1992

Exact Arithmetic Solid Modeling (Ph.D. Thesis)

Juixun, Yu

Report Number:
92-037

EXACT ARITHMETIC SOLID MODELING

Jiaxun Yu

CSD-TR 92-037
June 1992

EXACT ARTIHMETIC SOLID MODELING[1]

Jiaxun Yu Computer Science Department
Purdue University
West Lafayette, IN 47907
Technical Report CSD-TR-92-037
CAPO Report CER-92-18
June 1992

[1]In partial fulfillment of the requirements for the degree of Doctor of Philosophy

EXACT ARITHMETIC SOLID MODELING

A Thesis

Submitted to the Faculty

of

Purdue University

by

Jiaxun Yu

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 1991

To my parent, my wife Yan and my son Arthur

ACKNOWLEDGMENTS

I would like to acknowledge the great support and encouragement of my advisor, Professor Christoph M. Hoffmann. His insight, guidance and patience made this thesis possible. I would like to thank the other members of my graduate committee: Professors Robert E. Lynch, David C. Anderson and Samuel S. Wagstaff, for their time spent on discussing the problems studied in the thesis and on carefully reading the thesis. I am also indebted to Professor Kokichi Sugihara who spent generous amount of time to discuss the arithmetic precision problem with me while he was visiting Purdue.

Being thousands miles away from my home country, I would not be able to make this endeavor through without friendships from many friends of mine here at Purdue. While it is impossible to enumerate all of their names, I would like to give special thanks to: Danny Chen, Jong-Hong Chuang, Ching-Shoei Chiang for time spent discussing Chinese politics; Bill Bouma, for talking about Purdue basketball games; Andrew Royappa, for providing me help with Unix system utilities; and Kevin Kuehl, Neelam Jasuja, Jianhua Zhou, Pam Vermeer and Insung Ihm for time spent on talking various topics from religion to research.

Finally, I would express my most sincere thanks to my wife Yan for her patience and steadfast support. Her sacrifice of being a single working mother for the past year is the ultimate force to drive me out of this seemingly endless venture.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Yu, Jiaxun. Ph.D., Purdue University, December 1991. Exact Arithmetic Solid Modeling. Major Professor: Christoph M. Hoffmann.

Robustness in geometric computation is an important subject and it the topic of a variety of research by many people. Yet, to date, there is no known provably robust algorithm for performing Boolean operations on solids. The primary difficulty lies in performing arithmetic operations where fixed precision floating point numbers are employed to carry out operations that require infinite precision. Consequently, topological decisions based on the results of finite arithmetic operations are error prone. We study the robustness problem in the context of Boolean operations on solids by implementing a solid modeler that is capable of performing both rational arithmetic and floating point arithmetic. The algorithm has been implemented in identical code except for arithmetic. Therefore, it clearly demonstrates the effects of numerical errors on Boolean operations in those cases where the algorithm produces correct results with rational arithmetic but fails with floating point arithmetic. We analyze spatial configurations of solids that could result in failure of Boolean algorithms when floating point arithmetic is adopted.

With inevitable numerical errors in floating point arithmetic, it seems attractive to use rational arithmetic when implementing Boolean algorithms. However, as shown by the classification operations, this is feasible only when dealing with linear objects such as lines and planes. We study the precision required for exactly classifying a point, defined as the intersection of two lines or three planes, with respect to a given line or plane. Assuming line and plane equations have bounded integer coefficients, we need roughly four and five times of the input precision for point/line and point/plane

classification respectively and we also show that this result is optimal. Next we extend the concept of exact classification to the curve and surface domain. We study a resultant based method to exactly classify a point with respect to a given conic or quadric. The required precision is shown to be too high to be practical. Using piecewise linear approximations of conics and quadrics, the same problem is reduced to the exact point/line or point/plane classification problem which has previously solved. The required precision of the approximations is also analyzed.

1. INTRODUCTION

Boolean operations, that is, regularized intersection, union and difference, on solids play a fundamental role in solid modeling. They are used in various applications such as mechanical engineering, computer graphics, robotics and computer vision. Two representations are widely used. The boundary representation describes solids as a set of vertices, edges and faces and topological relations among them. In contrast, constructive solid geometry, or CSG, considers solids as expressions of Boolean operations and rigid-motion transformations of primitive solids which typically include block, sphere, cylinder, cone and torus.

Algorithms for performing Boolean operations have been proposed by many authors [31, 19, 38, 20, 17, 3, 12, 37]. In the early algorithms, little attention has been paid to numerical problems in the computation. Geometric data are usually represented by floating point numbers of fixed precision, and hence numerical errors in the computation are inevitable. In consequence, one had unexpected failures of the program implementing Boolean algorithms even though no algorithmic errors could be found.

It is now understood that the root cause of such failures of Boolean algorithms in the presence of floating point numbers is that these algorithms base their topological judgements on numerical computations that are vulnerable to numerical errors, such as round-off, cancellation and input perturbation. The problem of robustness for geometric algorithms, that is, to design robust geometric algorithms to cope with numerical errors arising from the use of finite precision arithmetic in the computation has become an important problem attracting quite a few researchers [4, 9, 12, 13, 15, 33, 22, 32, 2].

1.1 Effects of Numerical Errors in Geometric Computation

To see how numerical errors in geometric computation can affect the result of the geometric operation, let us look at the example of rotating a simple polygon around the origin. Assume that the vertex coordinates of the polygon are represented by floating point numbers with 4 bits mantissa and 3 bits exponent. Table 1.1 lists the vertex coordinates before rotation.

To rotate a point about the origin, we apply

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

where (x', y') is the rotated point coordinates. Let the rotation angle $\theta = 28°$. Then $\cos\theta = 0.1110$ and $\sin\theta = 0.1111 \times 2^{-001}$. Assume no accuracy loss in the intermediate computation and assume that the final results are rounded. The rotated point coordinates are listed in Table 1.2.

Now, let us first define the polygon as a triangle consisting of points P_1, P_2 and the origin O. After the rotation, the triangle becomes $P_1'P_2'O'$ where $O' = O$. It is easy to verify that the absolute values of the slopes of both lines OP_1 and OP_1' are greater than those of lines OP_2 and OP_2', respectively. If edges of the triangle OP_1P_2 are OP_1, P_1P_2 and P_2O and the convention is that the polygon area lies locally to the right of an edge, then OP_1P_2 encloses the finite area inside the bounding triangle while $OP_1'P_2'$ encloses the infinite area outside the triangle, see Figure 1.1.

If the polygon is picked as $P_1P_2P_3P_4$ which is a parallelogram, then the rotated polygon $P_1'P_4'P_3'P_2'$ is degenerate since P_1', P_2', P_3' and P_4' are now collinear residing on the same line $2x + y - 2 = 0$, see Figure 1.2.

Next, suppose $P_1P_2P_5P_6$ is taken. The result of rotation is a triangle since $P_5' = P_6'$, see Figure 1.3. Finally, it is not difficult to check that the rotated $P_1P_2P_3O$, $P_1'P_2'P_3'O$, has a self-intersection (Figure 1.4).

Because numerical errors may cause topological changes to the geometric objects during geometric computation, while the algorithm assumes that no alterations of

Table 1.1 Vertex coordinates before rotation

Point		Mantissa	Exponent	Binary	Decimal
P_1	x_1	+0.1001	+001	1.001	1.125
	y_1	+0.1110	+100	1110	14
P_2	x_2	+0.1010	+001	1.01	1.25
	y_2	+0.1111	+100	1111	15
P_3	x_3	+0.1010	+001	1.01	1.25
	y_3	+0.1100	+100	1100	12
P_4	x_4	+0.1001	+001	1.001	1.125
	y_4	+0.1011	+100	1011	11
P_5	x_5	+0.1010	+001	1.01	1.25
	y_5	+0.1000	+100	1000	8
P_6	x_6	+0.1001	+001	1.001	1.125
	y_6	+0.1000	+100	1000	8

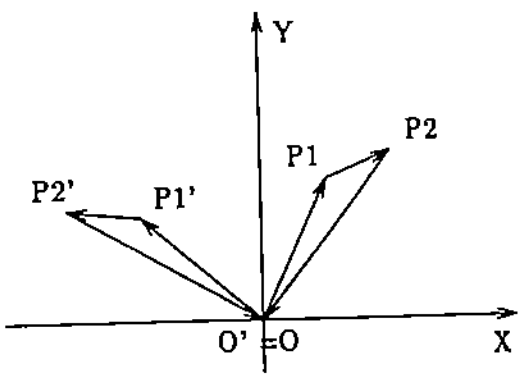

Figure 1.1 Triangle before and after rotation

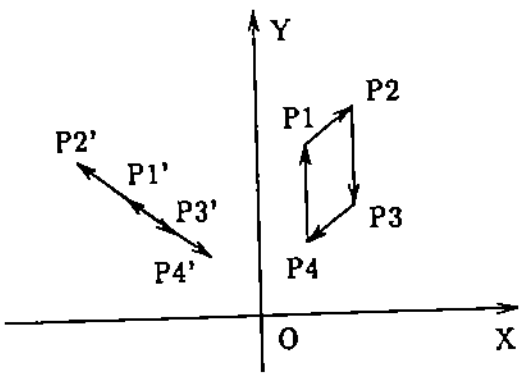

Figure 1.2 Degenerate polygon after rotation

Table 1.2 Vertex coordinates after rotation

Point		Before rounding	After rounding		Binary	Decimal
			Mantissa	Exponent		
P_1'	x_1'	-101.100101	-0.1011	+011	-101.1	-5.5
	y_1'	+1100.11000111	+0.1101	+100	1101	13
P_2'	x_2'	-101.11110	-0.1100	+011	-110	-6
	y_2'	+1101.1011011	+0.1110	+100	1110	14
P_3'	x_3'	-100.10001	-0.1001	+011	-100.1	-4.5
	y_3'	+1011.0001011	+0.1011	+100	1011	11
P_4'	x_4'	-100.001011	-0.1000	+011	-100	-4
	y_4'	+1010.00100111	0.1010	+100	1010	10
P_5'	x_5'	-10.10101	-0.1011	+010	-10.11	-2.75
	y_5'	+111.1001011	0.1111	+011	111.1	7.5
P_6'	x_6'	-10.110001	-0.1011	+010	-10.11	-2.75
	y_6'	+111.10000111	+0.1111	+011	111.1	7.5

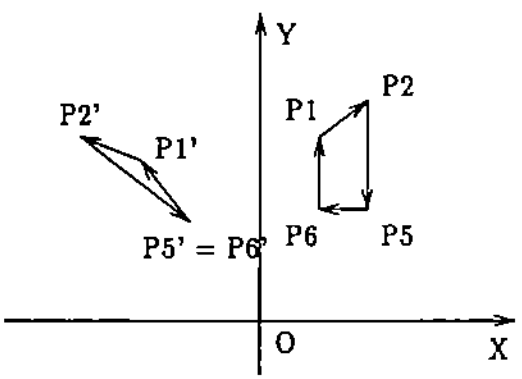

Figure 1.3 Coincident vertices after rotation

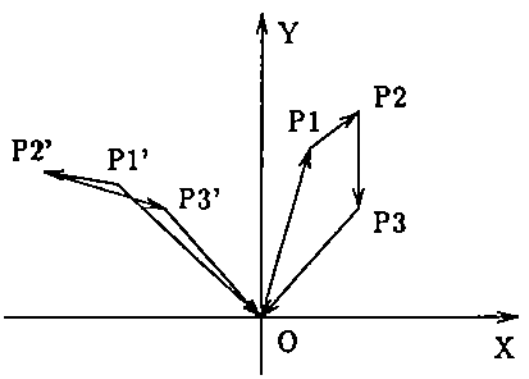

Figure 1.4 Self-intersection after rotation

topology can happen to the geometric objects, the program that implements the algorithm may either produce incorrect results or terminate abnormally when floating point numbers are used to represent the geometric data. Chapter 4 of the book by Hoffmann [12] contains a detailed account of the effects of numerical errors on geometric computation. A good example of how numerical errors can invalidate compounded simple geometric operations, also cited in Chapter 4 of [12], can be found in Dobkin and Silver [4].

1.2 Known Approaches for Solving the Robustness Problem

Most approaches found in the literature attempt to compensate for arithmetic errors by algorithmic steps which are sometimes heuristic. In contrast, Sugihara and Iri [33] propose exact rational arithmetic. We summerize major approaches to dealing with the robustness problem.

1.2.1 Exact Arithmetic

By observing the fact that the faces of a polyhedron obtained from Boolean operations always are in planes of the input polyhedra, Sugihara and Iri [33] take the plane equations as fundamental metric data in representing solids. Furthermore, assuming that the coefficients of the plane equations are in the appropriate ranges, the representable planes as well as their intersection points are finite in number. Hence, only finite precision is needed to decide the topology of the solids. To make fundamental metric data unique, Sugihara and Iri [33] require that any point be defined exactly by three intersecting planes and any plane of primitive polyhedra contains exactly three noncollinear points. They show that any topological judgement can be reduced to determining the sign of a 4×4 determinant and the required precision for the sign determination is five times the input precision. However, the result of the Boolean operations in this approach is a collection of planes that bound the volume of the output polyhedron. Explicit boundary representation of the result polyhedron

including vertex coordinates is not possible since we do not know the precision necessary to compute the vertex coordinates of the polyhedron. Moreover, coordinate transformations are permitted only on primitive solids where topological inconsistencies do not arise. This approach is important because it can decide the topological structure precisely.

1.2.2 Reasoning Paradigm

Hoffmann, Hopcroft and Karasick [15] introduce a reasoning method to cope with the incident/degenerate cases. Realizing that topological decisions are dependent, one wants to ensure that later decisions will not violate previous decisions. A set of rules and incidence tests have been devised in detail and are incorporated into the Boolean algorithm to guarantee that topological decisions are free from inconsistencies by Karasick [17]. It has not been proved that this algorithm solves the robustness problem completely. Following the same paradigm, Hopcroft and Kahn [16] show that for a class of problems, such as intersecting two convex polyhedra, algorithms that solve these problems can be structured in a way that topological decisions can be made in an independent manner. They assume, in the case of intersecting two convex polyhedra, that input polyhedra are given in the so called α-representation which means that the representation must satisfy certain conditions such as minimum feature separation. Furthermore, they assume there is sufficient precision for the intermediate computation. Then by utilizing the correspondence between equilibrium stressed graphs and convex polyhedra, they prove that a fairly straightforward algorithm of intersecting a convex polyhedron and a half space always produces a valid α-representation of the correct output convex polyhedron. However, they indicate that the same strategy does not extend to problems involving more complicated topological structures such as a nonconvex polyhedron.

1.2.3 ϵ Geometry

The method proposed by Guibas, Sales and Stolfi [9] assigns to each geometric predicate an uncertainty interval, called ϵ-interval. For tolerances outside the ϵ-interval, the predicate gives a definite answer. The method would work, at least, theoretically, for computations in any precision. However, it has limited usage due to the following reasons: (1) If the result of a predicate depends on complicated numerical computations, the explicit formula for the ϵ-interval will be too complex to be practically useful; (2) The calculation of the ϵ-interval usually over estimates errors and (3) This method could never show that a point is on a line — only that it is near the line.

1.2.4 Uncertainty Regions

Associating three regions with each primitive geometric object, such as, point and edge segment, Brüderlin [2] proposed a different method for solving the robustness problem. The tolerance region represents an upper bound of round-off errors in computing the primitive geometric object. Two primitive objects are said to be equal whenever their tolerance regions have nonempty intersection. The distinct region, on the other hand, differentiates two primitive objects whenever their corresponding distinct regions have empty intersection. Finally, the buffer region, defined as the intersection of the distinct regions of all the objects that are tested as equal, is used for computational convenience. To ensure that later topological decisions are consistent with previous ones, these three regions are updated whenever incidence tests are performed. Hence, topological decisions can be made consistent throughout if no 'ambiguous' configuration exists. The program incorporating the mechanism has to be rerun with a larger tolerance if any 'ambiguity' occurs in the computation. The approach makes it possible for Euclidean equality to be an equivalence relation in floating number computation at the price of increasing the size of the regions proportional to the number of equality tests. A major problem with this method is that when updating three regions in the incidence tests, numerical errors in computing

new regions have not been accounted for. Moreover, it is not clear whether such a scheme can be incorporated into geometric algorithms with constant time complexity.

1.3 About this Thesis

This thesis is about understanding and analyzing the robustness problem and makes an attempt to solve the problem. Although we are unable to propose a provably robust Boolean algorithm in finite precision, we hope that our findings add to the understanding of the nature of the problem so that better algorithms can be devised.

We study the robustness problem in two parts. On the algorithmic level, we investigate the problem by developing a solid modeler that implements a Boolean intersection algorithm in a way that permits evaluating experimentally heuristics for dealing with numerical failures. In order to distinguish algorithmic failure from numerical failure our modeler is capable of rational arithmetic as well as floating point arithmetic. Both modes of operation use identical code. Therefore, if an operation succeeds in rational mode but fails in floating point mode, we are assured that the failure is due entirely to floating point arithmetic.

Geometric algorithms, such as those for Boolean operations, execute simpler geometric operations. A fundamental operation is to classify a given point with respect to a solid, that is, to determine whether the point is in/out/on the other object. This operation, in turn, further reduces to more primitive operations such as classifying the point with respect to a line or a plane. This simpler classification will be referred to as the classification problem. In the second part of our research, we investigate the precision required to solve the classification problem by reducing it to the problem of finding the minimum distance between a given point and a given line, plane or surface. Knowing such minimum distance, not only are we able to achieve the precise classification as in the exact method, but we can also compute the intersection vertex coordinates.

Chapter 2 contains materials pertaining to Boolean operations. Representation of solids, data structure and a Boolean intersection algorithm are described. Chapter 3 summarizes an empirical study of categorizing possible failures of the Boolean algorithm described in the Chapter 2. We provide analysis to show the cause of the failures and propose possible heuristic remedies for the failures. Chapter 4 estimates the precisions required for the point/line and point/plane classification problems. Combining the results from Sugihara and Iri [33], lower bounds on the precision required for 2-D point/line and 3-D point/plane classifications are derived. Chapter 5 describes a method that can be used to exactly classify point/conic relationships. Finally, Chapter 6 concludes the thesis with some remarks.

2. THE MODELER AND BOOLEAN OPERATIONS

In order to determine whether numerical errors or algorithmic errors cause failures of Boolean algorithms in a solid modeler, we developed our experimental solid modeler. The input to the modeler is a list of vertex coordinates and a list of faces defined by bounding vertices given in clockwise order in the plane that containing the face. The modeler has the following unique feature. It does the same analysis for input vertex coordinates in rational numbers or in floating point numbers. Which arithmetic mode is chosen depends on the input data. If the input vertex coordinates are rational numbers, then every internal arithmetic operation is done in rational arithmetic. If the input vertex coordinates are floating point numbers, then every internal arithmetic operation is accordingly done in floating point arithmetic. So, if we can discover any case in which the modeler succeeds for rational vertex coordinates, but fails for the same objects input with floating point vertex coordinates, the failure would be evidence that the numerical errors during the floating point computation prevented the modeler from working properly.

In the following sections, we describe our algorithm in detail starting with some necessary background. Since we place our emphasis on the robustness issue, our description of basic concepts is informal. We recommend [12, 20] for a formal treatment on the subject.

2.1 Input Solid Domain

Solids considered in this thesis are 3-D polyhedra which contain 3-D volumes in the space. Each face of a polyhedron is a bounded planar polygon that has no self-intersection. Each polygon may consist of several connected components each of which is called a loop. 3-D polyhedra are usually classified into two classes: manifold

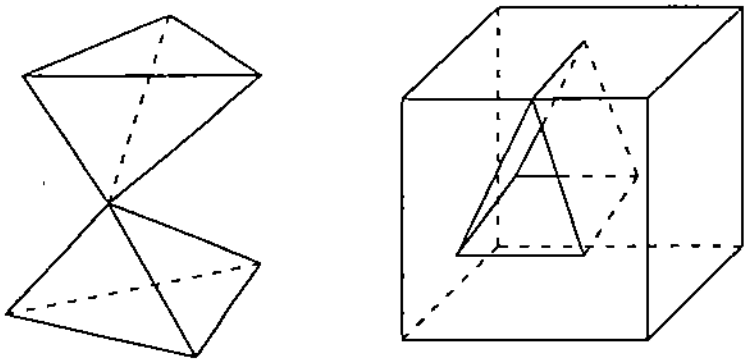

Figure 2.1 Nonmanifold solids

polyhedra and nonmanifold polyhedra. Informally speaking, manifold polyhedra are those in which each edge is adjacent to exactly two faces and there is exactly one cone of alternating faces and edges at every vertex. Nonmanifolds polyhedra, on the other hand, permit more than two, but always an even number of, faces to be associated with an edge and more than one cone at a vertex. Examples of nonmanifold solid are shown in Figure 2.1.

Traditionally, manifold polyhedra received more attention than nonmanifold polyhedra. The reason is of twofold: On one hand, compact representations exist for manifold polyhedra due to its well understood topological properties. Hence, Boolean operations on manifold polyhedra can be done more efficiently. On the other hand, manufacturable mechanical parts which may be designed by a solid modeler are manifold polyhedra. But, manifold polyhedra are not closed under Boolean operations. Figure 2.2 shows the union of two cubes that is a nonmanifold polyhedron.

To allow uniform treatment of polyhedra both as input and as output, we choose as domain of solids for our Boolean operations nonmanifold polyhedra.

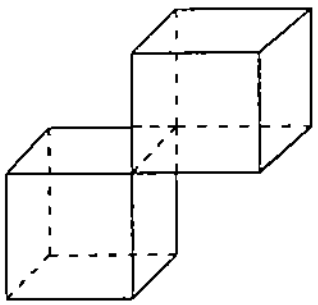

Figure 2.2 Union of two cubes

2.2 Regularized Boolean Operations

The Boolean operations on 3-D polyhedra in the set-theoretic sense may produce output that is not homogeneously three-dimensional. Typical examples of lower dimensional geometric entities in the output solid are dangling faces. To ensure that a solid contains a volume of the 3-D space, we use regularized Boolean operations. The regularization of a point set that contains a 3-D volume is the closure of the interior of the point set. Applying regularization to a solid eliminates all the lower dimensional structures. From now on, we assume that Boolean operations are always regularized Boolean operations. $\cap^*$ and $\cup^*$ stand for regularized intersection and regularized union respectively.

2.3 Representation Scheme

We represent solids in this thesis by enumerating vertices, edges and faces of the solid boundary. This is called the boundary representation of a solid. The topological aspect of the solid is reflected by recording the adjacency and incidence relationships among the vertices, edges and faces. The geometric data of the solid consists of coordinates of the vertices and plane equations of the faces.

Elegant boundary representations were proposed in the past for representing manifold polyhedra. The notable examples are the Winged-Edge representation by Baumgart [1], the Quad-Edge representation by Guibas and Stolfi [10], and the Winged-Triangle representation by Paoluzzi et al. [24]. However, such compact and topologically complete representations are not available to represent nonmanifold polyhedra. The difference is that in a manifold polyhedron the adjacency information at an edge and a vertex is fixed. There are always two faces incident to an edge and there is only one cone — a cycle of alternating edges and faces — associated with a vertex. But, in a nonmanifold polyhedron, there can be $2n$ faces, where $n > 1$, incident to an edge and there can be more than one cone at a vertex. The key ingredient in representing nonmanifold polyhedra is that one has to make a distinction between an edge being used when identifying a face, which we call the edge-use of the edge on that face, and the edge itself, which stores its spatial position. With edge-use to represent each face, we can account for an important consideration — separating the topological data from the geometric data of an edge, a concept first introduced by Weiler [38], and subsequently adopted and called directed edge by Karasick [17] in his Star-Edge representation and called fedge by Vaněček [37] in his Fedge representation. Apart from the adoption of the concept of edge-use, most of the existing representations for representing nonmanifold polyhedra are essentially the same. There can be some minor differences in detail from one representation to another depending on the specific algorithm and efficiency considerations. For surveys on solid representations, readers are referred to the relevant chapters of Weiler [38], Karasick [17], Vaněček [37] and Hoffmann [12].

The representation used in our Boolean algorithm follows the one that has been described in section 3.2 of [12] with some extra fields. The extra fields, which will be described in detail, are used primarily for robustness consideration. Overall, our representation also resembles Karasick's Star-Edge representation. The representation hierarchy is shown in Figure 2.3 with each box representing an entity, arrows indicating one way pointers and lines indicating pointers in both directions.

We assume that a polyhedron has faces that enclose a finite area but may enclose infinite volume. The reason for allowing polyhedra with infinite volume is to eliminate the need of implementing a separate polyhedra union algorithm. The union of two polyhedra, according to de Morgan's law, can then be converted by applying the complement operation to the result of intersecting the two complemented input polyhedra. For each face, we assume that the area of the face lies locally to the right of an edge-use and the normal of the plane containing the face points to the exterior of the solid. In what follows, we describe our representation in detail.

2.3.1 Solid and Shell

The top level structure — solid, consists of a list of shells, a list of vertices and a list of edges of the solid. A boolean flag field called universe is used to indicate whether the solid is the entire space or the empty space when there are no vertices, edges and shells. A shell is a list of connected adjacent vertices, edges and faces. Topologically, a shell is a maximal connected set of vertices, edges and faces bounding the solid. The representation for a shell includes a back pointer to the solid that the shell belongs to and the list of faces bounding the shell. Infvol is a boolean flag field indicating whether the shell encloses finite or infinite volume. An example of a shell having infinite volume would be an internal void inside a cube. We do not organize the shells of the solid into a tree according to their spatial containment relation. Rather, we keep all the shells in a list. This entails a little more analysis when testing the containment relationship of two shells but saves the work of building a shell tree. Finally, inted is another boolean field indicating whether the shell has any intersection with shells of the other solid. When two shells do not intersect, we should apply a shell containment test on them to determine the shell(s) that will be included in the intersecting polyhedra.

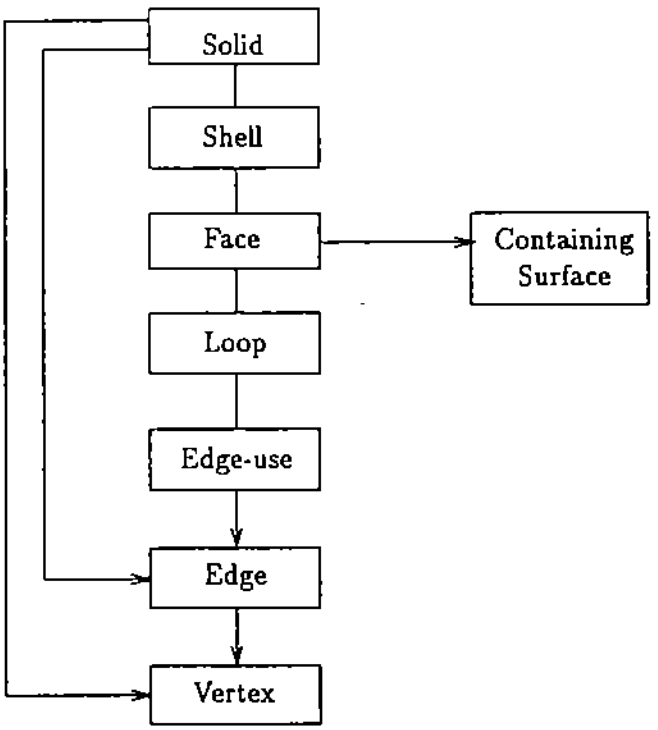

Figure 2.3 Hierarchical solid representation

Table 2.1 Solid representation

FIELD
universe
vertices
edges
shells

Table 2.2 Shell representation

FIELD
psolid
faces
infvol
inted

Table 2.3 Face representation

FIELD
pshell
peqn
orient
loops
mark

Table 2.4 Loop representation

FIELD
pface
type
pointer
orient
inted

2.3.2 Face and Loop

A face consists of a list of loops. Each loop is a connected list of edge-uses bounding that face. Each face has a back pointer to the shell the face belongs to and has a pointer to the structure PlaneEqn where the equation of the face is stored. We store the equation of the plane containing the face separately from the face itself since the plane equation is considered to be a geometric datum which may be shared by several different faces possibly with different orientations. By assumption the equation of the plane which contains a face has its normal pointing towards the exterior of the solid, so each face has a boolean field, orientation, indicating how to orient the plane equation so that the assumption is met. Mark is a scratch boolean field used for the following purpose: after having assembled and collected intersection faces between two solids, we need to perform a depth first search on all these intersection faces to locate all the adjacent faces of each intersecting shell. After one intersection face has been explored, we set mark to be true.

A loop generally is a connected list of edge-uses, or directed edges by some authors, bounding the area of the face to which it belongs. The representation of a loop contains a back pointer to the face so that information about face can be accessed. By convention face area lies to the right of an edge-use, so each loop has a boolean field called orientation to indicate whether the loop encloses area or hole of the face. There

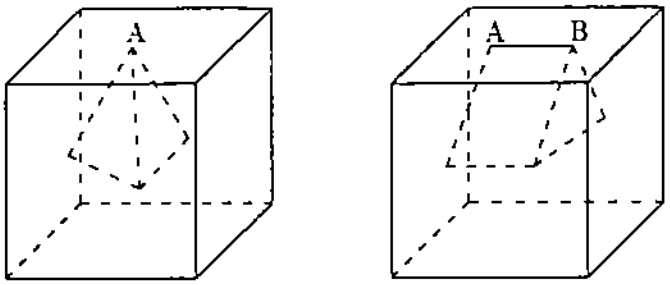

Figure 2.4 Singular loops

can be two types of singular loops: an isolated vertex and a connected graph of edge-uses enclosing zero area. The cases where singular loops arise are shown in Figure 2.4. The type of the loop is recorded in the field called type and the field pointer will point to a vertex if the loop is an isolated vertex or an edge-use otherwise. Note that there may be two edge-uses with opposite direction but sharing the same edge in a loop. One such instance is the only loop of the top face of the right solid shown in Figure 2.1. Similar to representing a list of shells, we do not organize the list of loops of the face into a tree structure by their area containment relationship. When intersecting two coincident faces (the same plane contains both faces) of the two solids, we have to record whether these loops intersect each other or not. This information is kept in the field called inted and will be used to test containment relationship of each pair of nonintersecting loops of the two faces in order to generate the final intersection face. This is exactly the 2-D counterpart of the shell containment test in 3-D.

2.3.3 Edge-use

An edge-use is an instance of the edge being used in a specific loop of a face. An edge-use has a back pointer to the loop it resides and it also has a pointer to the physical edge that the edge-use is on. Each edge-use is directed such that when moving from its start to its end vertex, the area of the face lies to the right of the

edge-use. Since the edge to which the edge-use points has a default direction, a boolean field, orientation, is used to keep the information of whether the edge-use has the same or opposite direction as the default edge direction. Using the field next, the edge-uses of a loop are connected into a circular linked list. At each vertex of the face, edge-uses are radially sorted in clockwise order. After sorting, we pair each edge-use directed away from the vertex with the next edge-use in the sorted order which must be an edge-use directing towards the vertex. Each of these pairs must enclose face area and is thus called an area-enclosing pair. The field areapair is used to link the two edge-uses of the same area-enclosing pair.

We associate each edge-use with a vector called face direction vector which points into the interior of the face area and is perpendicular to the edge-use. We use fdirect to record the face direction vector of the edge-use. Fdirect is computed by

$$\text{Direct(edge-use)} \times \text{FaceNormal(face)}$$

where Direct(edge-use) computes the edge-use direction vector, see Figure 2.5. Note that if there are two opposite directed edge-uses sharing the same edge on a face, then the two face direction vectors associated with them are in opposite direction, see the right picture of Figure 2.5. After the face direction vector is defined for each edge-use, then faces adjacent to each edge can be sorted radially in clockwise order according to the face direction vector of each edge-use associated with each incident face. Similar to pairing area-enclosing pairs in a face, we now pair consecutive faces into wedges enclosing volume of the solid and call each pair a volume-enclosing pair. The field volpair is used to link the two edge-uses of a volume enclosing pair. Finally, edge-gened is a scratch field that will be set to be true when the corresponding edge of the edge-use has been created while traversing the intersection solid.

2.3.4 Edge

An edge has two bounding vertices. We assign arbitrarily the start vertex to svertex and the end vertex to evertex and establish a default orientation vdirect of

Table 2.5 Edge-use representation

FIELD
ploop
orient
pedge
fdirect
next
areapair
volpair
edge-gened

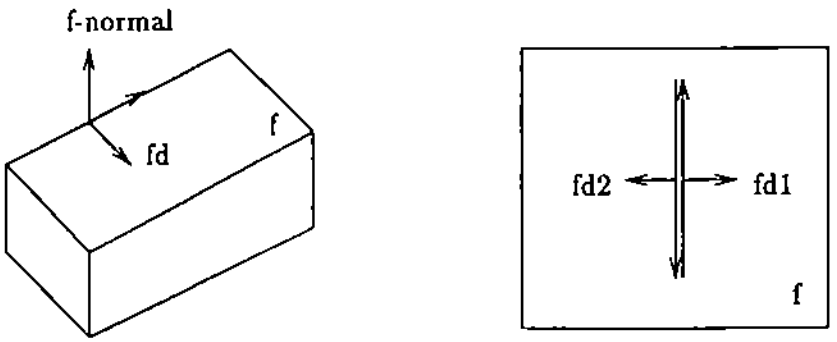

Figure 2.5 Face direction vector of an edge-use

Table 2.6 Edge representation

FIELD
svertex
evertex
vdirect
adjacent
refedge
subdiv
sorted
intflist
intelist
intedge
ftrans

the edge as

Coord(evertex) - Coord(svertex)

where Coord(vertex) accesses the coordinates of the vertex. The adjacency of an edge is a list where each element of the list is a pair consisting of an incident face and the edge-use on that face. The list is radially sorted in clockwise order in a plane with vdirect as normal and paired into volume-enclosing pairs as described earlier. Each edge may be subdivided when there is one or more faces of the other solid that intersect the edge in its interior. However, we do not create subdividing edge segments each time when the edge is intersected by a face of the other solid. Rather, we keep all subdividing intersections in a list and store it in the field subdiv. After all the pairs of faces between the two solids are intersected, we sort the list of subdividing intersections of each edge along the default edge direction. The traversing algorithm can then pair the proper intersections into edges of the intersection solid. Sorted is the related boolean field indicating whether the list of subdividing intersections of the edge have been sorted or not when a traversing algorithm traverses the edge.

From the robustness point of view, each geometric quantity ought to be computed only once to avoid possible inconsistency that may arise from multiple computation of the same geometric quantity. The Boolean algorithm intersects all pairs of faces of the two solids. An intersection between an edge of the first solid and the face of the second solid is discovered when intersecting the corresponding edge-use of the first face and the face of the second solid. The same intersection may be rediscovered later when intersecting adjacent faces of the edge and the other face. In order to avoid computing the same intersection point repeatedly, we have to record the information whenever an intersection between an edge and a face is found. This is the purpose of having the field intflist and intelist. We put a pair of the face and the intersection point in the edge's intflist when the intersection is first computed. Subsequently, when intersecting an edge-use and a face, the intflist of the edge that is referenced by the edge-use is looked up first to see whether the edge and the face has already been intersected before actually computing the intersection. If the edge intersects the

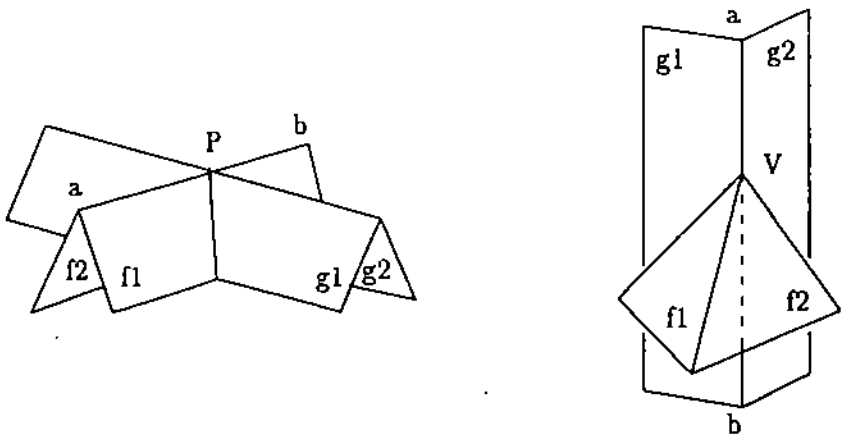

Figure 2.6 Transfer edge/edge and edge/vertex intersection

other face at a point that is interior to both the edge and the face (the case when one of the vertices of the edge is on the face will be dealt at the vertex representation), the information in the edge's intflist is sufficient to avoid any unwanted recomputation of the intersection point. However, if the edge intersects the edge or vertex of the other face, then the above simple book keeping may not be enough to avoid all the unwanted intersection computations.

For example, consider the left picture in Figure 2.6, the intersection point P was discovered when intersecting face f_1 and g_1 and then P is stored in edge ab's intflist. So when face f_2 intersects g_1, P is retrieved from ab's intflist. But when intersecting f_1 and g_2, P may be recomputed from ab and g_2 since g_2 does not know that ab intersected an edge that g_2 is adjacent to. Similar problems exist in the right picture of Figure 2.6 when ab intersects a face, say f_1, at a vertex V. To correct the above problems, we need to propagate intersection information not only to faces adjacent to edge ab but also faces incident at the edge or vertex that ab intersects. We call such propagation as intersection transferring. The following algorithms transfers edge/edge intersection information. Note that ftrans and intedge are related book keeping fields used by transferring algorithms.

```
procedure TRANSFER.EDGE.EDGE(u, v, p)
/* transfer intersection p to faces adjacent to edge v */
begin
    if v is not in u's ftrans then
        for each face f adjacent to v do
            push (f, p) into u's intflist;
        endfor;
        set u's intedge to v;
        push v into u's ftrans;
    endif;
end
```

Algorithm: Transfer edge/edge intersection

Input: edge e_1, e_2 and their intersection int;

call TRANSFER.EDGE.EDGE(e_1, e_2, int);

call TRANSFER.EDGE.EDGE(e_2, e_1, int);

When edge/vertex intersection occurs, then the intersection information must be transferred to both faces and edges adjacent to the vertex. We use the field intelist to store a paired list where each pair is an edge adjacent to the vertex and the vertex intersection. Note that if the edge intersects the face transversally, then intflist is sufficient to infer all the transferred intersection information. Intelist is useful when the edge is on the face since we then have to perform a planar edge and polygon intersection which will utilize the information to avoid unwanted planar edge and edge intersection. After the intersection transferred to the adjacency of the vertex, we also transfer the information to the adjacent faces of the edge. The transferred information is stored in the vertex's fields intflist and intelist respectively.

```
procedure TRANSFER.EDGE.VERTEX(u, v, p)
/* transfer p to faces and edges adjacent to vertex v */
begin
    if v is not in u's ftrans then
        for each face f adjacent to v do
            push (f, p) into u's intflist;
        endfor;
        for each edge e adjacent to v do
            push (f, p) into u's intelist;
        endfor;
        push v into u's ftrans;
    endif;
end

procedure TRANSFER.VERTEX.EDGE(u, v, p)
/* transfer intersection p to faces adjacent to edge v */
begin
    if v is not in u's ftrans then
        for each face f adjacent to v do
            push (f, p) into u's intflist;
            push (f, 0) into u's fevallist;
        endfor;
        push (v, p) into u's intelist;
        for each edge e adjacent to u do
            if v is not in e's ftrans do
                for each face f adjacent to v do
                    push (f, p) into e's intflist;
                endfor;
                push (v, p) into e's intelist;
```

```
                endif;
            endfor;
            push v into u's ftrans;
        endif;
    end
```

Algorithm: Transfer edge/vertex intersection

Input: edge e, vertex v and their intersection int;

call TRANSFER.EDGE.VERTEX(e, v, int);

call TRANSFER.VERTEX.EDGE(v, e, int);

Note that the procedure TRANSFER.VERTEX.EDGE used a field called fevallist in the vertex structure. The role of the field will be explained shortly in the vertex representation.

2.3.5 Vertex

Geometrically, a vertex is a point in the 3-D space. The coordinates of the vertex is stored in coord as a list of x, y and z coordinates. Topologically, a vertex has a neighboring structure of a set of cones incident to the vertex. Information about adjacent edges and faces is stored in adjacent. The adjacency structure is a nested list where each element of the list is itself a list indexed by an adjacent face. The element list consists of the index face and subsequently the list of radially sorted area-enclosing pairs incident at the vertex. Using Backus-Naur Form notation, the adjacency can be represented as

$$\{\ face\ \{\ area\text{-}enclosing\ pair\ \}+\ \}+$$

Intflist, intelist and ftrans have the same meaning as explained in edge representation and are used in transferring intersection information to adjacent entities in the following situations:

1. The vertex is on a face of the other solid: Transfer intersection information to all the edges adjacent to the vertex to ensure that each edge intersects the face at the same vertex intersection;

2. The vertex is on an edge of the other solid: Apply the same procedures as those of transferring edge/vertex intersection described in the edge representation;

3. The vertex is on a vertex of the other solid: We use the following algorithm to transfer vertex/vertex intersection.

```
procedure TRANSFER.VERTEX.VERTEX(u, v, p)
/* transfer intersection p to faces and edges adjacent to vertex v */
begin
    if v is not in u's ftrans then
        for each face f adjacent to v do
            push (f, p) into u's intflist;
            push (f, 0) into u's fevallist;
        endfor;
        for each edge e adjacent to v do
            push (e, p) into u's intelist;
        endfor;
        for each edge e adjacent to u do
            if v is not in e's ftrans do
                for each face f adjacent to v do
                    push (f, p) into e's intflist;
                endfor;
                for each edge e adjacent to v do
                    push (e, p) into e's intelist;
                endfor;
            endif;
        endfor;
```

```
            push v into u's ftrans;
        endif;
    end
```

Algorithm: Transfer vertex/vertex intersection

Input: vertex v_1, v_2 and their intersection int;

```
call TRANSFER.VERTEX.VERTEX(v1, v2, int);
call TRANSFER.VERTEX.VERTEX(v2, v1, int);
```

The field fevallist is used in computing the coordinates of the intersection point of an edge segment and a face. Let $\overline{P_i} = (x_i, y_i, z_i, 1)$, $i = 1, 2$, be the homogeneous coordinates of the end points of the edge. The line passing through $\overline{P_1}$ and $\overline{P_2}$ is $\overline{l(t)} = \overline{P_1} + t(\overline{P_2} - \overline{P_1})$. The plane equation of the face can be written as $N \cdot \overline{P} = 0$ where $N = (a, b, c, d)$ and $\overline{P} = (x, y, z, 1)$. The value t of the intersection point between P_1P_2 and the plane is determined from $N \cdot (\overline{P_1} + t(\overline{P_2} - \overline{P_1})) = 0$. So

$$t = \frac{N \cdot \overline{P_1}}{N \cdot \overline{P_1} - N \cdot \overline{P_2}}$$

To avoid repeated evaluation, we store a pair of the face and its $N \cdot \overline{P}$ in the vertex's fevallist whenever the vertex has been evaluated against the plane equation that contains the face. The sign of $N \cdot \overline{P}$ classifies whether the vertex is above/on/below the face. An transversal intersection occurs when $N \cdot \overline{P_1}$ and $N \cdot \overline{P_2}$ are of opposite sign. By using intersection transferring and the above fevallist, we make sure that every intersection is computed only once and is unique throughout the algorithm.

After an intersection is discovered and properly transferred, we have to analyze its neighborhood. For example, in Figure 2.7, a vertex/face intersection is found and v is on face f. We use face f to clip away those edge-uses incident to v whose edges are above face f. After clipping, only faces and edge-uses incident to vv_1 and vv_2 are kept for further consideration. In case an edge e is on face f, then the edge will be

Table 2.7 Vertex representation

FIELD
coord
adjacent
fevallist
intflist
intelist
ftrans
int
copy
travlist
clip
mark

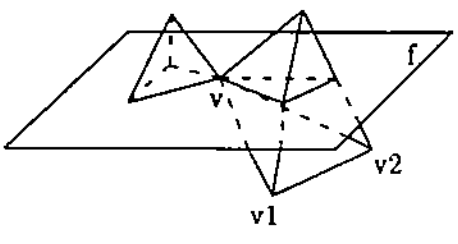

Figure 2.7 Neighborhood analysis for vertex/face intersection

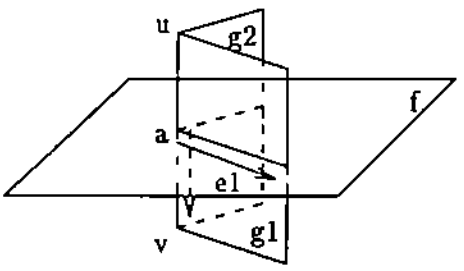

Figure 2.8 Traversing edge-use for intersection vertex

kept if face f splits at least one volume-enclosing pair of the edge e. We use clip to record the list of faces that have been used as clipping faces.

Upon completion of the analysis, the adjacency information has been established for each intersection vertex. We organize the adjacency information into a nested list indexed by face and store it in the field travlist. Each element of the list is a list consisting of a face and radially sorted out-going edge-uses on that face. The adjacency information at the time is only partially complete since some out-going edge-uses are from input solids. For example, in Figure 2.8, intersection vertex a has out-going edge-use e_1 on face g_1. We call e_1 good since the edge that e_1 corresponds to is an intersection edge. But on g_2, the out-going edge-use is uv which is from an input solid. Therefore, we will apply a traversing procedure later to travel along the travlist of each intersection vertex and trace out the intersection solid. The traversing procedure will create new edges, edge-uses and vertices when necessary to complete the adjacency information for vertices of the intersection solid.

Finally, if a vertex is an intersection itself, we create a new intersection vertex and use int to point to the original vertex. Copy and mark are boolean scratch fields.

2.3.6 Basic Geometric Operations

We describe in this section some basic geometric operations that are used repeatedly by the Boolean intersection algorithm. From the implementation point of view, our Boolean intersection algorithm is built on top of these basic operations.

2.3.6.1 Sort Planar Vectors Radially at a Point

When forming area-enclosing pairs at a vertex or forming volume-enclosing pairs at an edge, we need to sort radially a list of vectors at a point. This sorting is a two dimensional operation. To form area-enclosing pairs, we sort all the edge-uses that are in the plane and adjacent to the vertex. To form volume-enclosing pairs, we sort all the face direction vectors that are incident to the edge in a plane that is perpendicular to the edge.

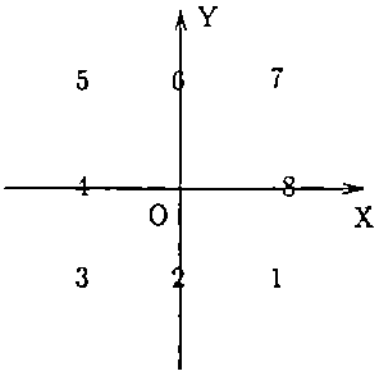

Figure 2.9 Assign number to quadrants and axes

To order these vectors, we have to establish a local coordinate system. When sorting edge-uses that are in a plane and adjacent to the vertex to form area-enclosing pairs, we choose the vertex as the origin. When sorting face direction vectors that are incident to an edge to form volume-enclosing pairs, we choose either the starting vertex of the edge or the ending vertex of the edge as the origin. Next we arbitrarily choose a vector X as the x-axis. Then the y-axis is determined by $N \times X$ where N is normal vector of the plane. We assign a number, called total quadrant, to the four quadrants and axes in clockwise order as shown in Figure 2.9. Given a vector v in the plane. The signs of $v \cdot x$ and $v \cdot y$ are sufficient to determine the total quadrant in which the vector v is. When vectors u and v are both in total quadrant 1,3,5 and 7, we use cos function to compare them. We define

$$\cos \alpha = \frac{u \cdot X}{||u||\,||X||} \text{ and } \cos \beta = \frac{v \cdot X}{||v||\,||X||}$$

When u and v are in the same total quadrant, $\cos \alpha$ and $\cos \beta$ have the same sign. To avoid computing square roots associated with $||u||$, $||v||$ and $||X||$, we compare $\cos^2 \alpha$ and $\cos^2 \beta$ instead.

$$u \text{ is radially greater than } v \Longleftrightarrow \begin{cases} \cos^2 \alpha < \cos^2 \beta & \text{if } u, v \text{ are in total quadrant 1, 5} \\ \cos^2 \alpha > \cos^2 \beta & \text{if } u, v \text{ are in total quadrant 3, 7} \end{cases}$$

or equivalently

$$u \text{ is radially greater than } v \Longleftrightarrow \begin{cases} \frac{(u\cdot X)^2}{u\cdot u} < \frac{(v\cdot X)^2}{v\cdot v} & \text{if } u, v \text{ are in total quadrant 1, 5} \\ \frac{(u\cdot X)^2}{u\cdot u} > \frac{(v\cdot X)^2}{v\cdot v} & \text{if } u, v \text{ are in total quadrant 3, 7} \end{cases}$$

The following algorithm radially orders two vector u and v.

Algorithm: Radially order two vectors at a point

Input: vector u, v in the local coordinate system;

Return: 1: if u is radially greater than v; -1: if u is radially less than v;

0: if u is radially equal to v;

compute u and v's total quadrant number;

let n=sign(u's number $-$ v's number);

if n is not zero then

return n;

else /* u and v have the same total quadrant number */

case u's total quadrant number do

1,5) return $-\text{sign}((v\cdot v)(u\cdot X)^2 - (u\cdot u)(v\cdot X)^2)$;

3,7) return $\text{sign}((v\cdot v)(u\cdot X)^2 - (u\cdot u)(v\cdot X)^2)$;

default) return 0;

endcase;

endif;

2.3.6.2 Pair Intersection Points into Intersection Edges

After intersecting one face with the plane containing the other face transversally, we obtain a list of intersection points. We describe here how to pair these intersection points into intersection edge segments or isolated vertices.

Assume that the intersecting faces are f and g and their corresponding planes are P and Q. The intersection points of g and P must lie on the line l that is the intersection of P and Q. Before pairing them, we have to sort all the intersection

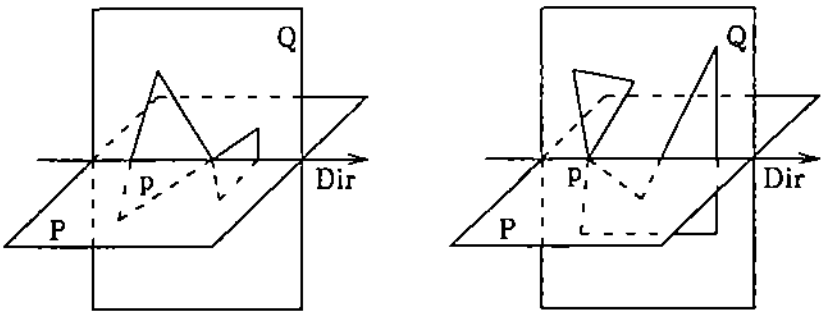

Figure 2.10 Pairing intersection points

points along the the intersection line l. The sorting direction is determined as follows:

$$\text{Dir} = \begin{cases} N_Q \times N_P & \text{if the shell } f \text{ belongs to is finite} \\ N_P \times N_Q & \text{if the shell } f \text{ belongs to is infinite} \end{cases}$$

Having computed Dir, we sort the list of intersection points by the coordinate which has the largest absolute value in Dir and then pair the ordered intersection points. We analyze at each intersection point p whether p is a cross intersection (i.e. a face and an edge intersect in the edge interior) or a vertex intersection. If p is a cross intersection and the line segment connecting p with the next point in sorted order is in the interior of face g, then p is paired with the subsequent point. If p is a vertex intersection, then depending upon whether Dir or $-$Dir splits an area-enclosing pair at p, we pair p with either preceding point or succeeding point or both. When neither Dir nor $-$Dir splits an area-enclosing pair at p but there exists an area-enclosing pair below the plane P, then p is an isolated vertex, see Figure 2.10.

Algorithm: Pairing intersection points into edges and isolated vertex
Input: a list of transversal intersection points;
Return: a list of paired intersection edge and isolated vertices;

creat_edge=false;

for each intersection point p do

```
        if p is a cross intersection do
            if creat_edge is true do
                pair pp with p;
            endif;
            creat_edge=not creat_edge;
        endif;
        if p is a vertex intersection do
            if −Dir splits an area-enclósing pair at p do
                pair pp with p;
            endif;
            if Dir splits an area-enclosing pair at p then
                creat_edge=true;
            else
                creat_edge=false;
            endif;
            if neither Dir nor −Dir splits an area-enclosing pair at p
              and there exists an area-enclosing pair below P do
                p is an isolated vertex;
            endif;
        endif;
        pp=p;
    endfor;
```

2.4 Boolean Intersection Algorithm

The algorithm is similar to the one that has been described in section 3.4 of the book by Hoffmann [12].

2.4.1 Global Description

The algorithm first intersects shells of solid A with shells of solid B. After analyzing all the intersection points and properly transferring intersection information, the algorithm assembles all the intersection shells of the intersection solid. If there exist nonintersecting shells, then the algorithm applies proper containment tests to these shells and collects those shells of solid A that are contained entirely in the interior of solid B and vice versa. Conceptually, the intersection algorithm is as follows:

1. For each face f of solid A, intersect each plane Q that contains a face g of solid B. Pair the list of intersection points into intersection edge segments and isolated vertices. Intersect g with the line l representing the intersection of the plane P containing f and plane Q to obtain the second set of edge segments. Then the two sets of edge segments are intersected to obtain the edges of the intersection solids. While constructing intersection edges, intersection information is properly transferred and traverse edge-uses are determined for each end point of an intersection edge.

2. If f and g are coplanar, then perform a two dimensional polygon intersection to obtain the intersection face. Loop containment tests are necessary to obtain the loops that do not intersect any loop of the other face but are in the interior of the other face.

3. Traverse each intersection point along out-going edge-uses stored in travlist to assemble faces of the intersection shells. This traversal will find all the faces of A that have intersection points with B and are in the interior of B and all the faces of B that have intersection points with A and are in the interior of A.

4. Finally, if there are shells of A that do not intersect B or shells of B that do not intersect A, then perform a three dimensional containment test to include those shells of A that are in the interior of B and shells of B that are in the interior of A.

2.4.2 Intersecting Two Transversal Faces

Computing intersection of two faces is one of the most essential parts of any Boolean intersection algorithm since computing intersection of two solids or two shells will eventually reduce to computing the intersection of two faces. Assume f and g are the two faces and P and Q are the planes that contain f and g respectively. The following algorithm details how to intersect two faces.

1. Intersect each edge-use of g with P and thus obtain a list of intersection points lying on the line l that is the intersection of P and Q.

2. Sort the intersection points along line l and pair them into a set of edge segments I_1 by the previously described algorithm.

3. Intersect f with l and obtain the second set of edge segments I_2 lying on the line l. Make sure both sets of edge segments are oriented in the same direction along line l.

4. Intersect each edge segment or isolated vertex of I_1 with each edge segment or isolated vertex of I_2. Assume that e_1 and e_2 are from I_1 and I_2 respectively. If e_1 is before e_2 along line l, then e_1 does not intersect e_2. Discard e_1 and assign e_1 the next element in I_1. Similarly, advance e_2 to the next element in I_2 if e_2 is before e_1 along line l. If e_1 intersects e_2, then there are four cases according to the types of e_1 and e_2: e_1 and e_2 are both edge segments; e_1 is an edge segment and e_2 is an isolated vertex; e_1 is an isolated vertex and e_2 is an edge segment; e_1 and e_2 are both isolated vertices. In each case, there may be an edge or isolated vertex of the intersection solid depending upon the three dimensional neighborhood analysis at these intersection points. We describe each case in detail in the next subsections.

5. Finally, a set of edges and isolated vertices of the intersection solid are obtained as the result of two face intersection.

Note that in step 1 of the algorithm, when intersecting an edge-use and a plane, we first look up the corresponding edge's intflist to find out whether the edge has intersected the face or not and compute the intersection point only if they do not intersect before. After computing a new intersection point, it will be stored in the edge's subdiv indicating that the edge should be subdivided at the point and a pair of the face and the intersection point will be stored in the edge's intflist.

The sorting and pairing algorithm for step 2 has been described earlier. Step 3 can be done similarly to step 2. In step 4, we transfer intersection information at each intersection point to adjacency structures to ensure topological consistencies. An edge or an isolated vertex of the intersection solid is obtained after applying three dimensional neighborhood analysis to the intersection points. Traverse edge-uses on the two faces will be properly determined for each end point of an edge or each isolated vertex of the intersection solid.

2.4.2.1 Generic Intersections

Generically, for two transversal intersecting faces, there are following intersection configurations among vertices, edges and faces.

1. Edge/face intersection: an edge-use intersects the other face at a point that is interior to both edge-use and face.

2. Edge/edge intersection: an edge-use intersects an edge-use of the other face at a point that is interior to both edge-uses.

3. Edge/vertex intersection: An edge-use intersects a vertex of the other face at a point that is interior to the edge-use.

4. Vertex/face intersection: a vertex intersects the other face at a point that is interior to the other face.

5. Vertex/edge intersection: a vertex intersects an edge-use of the other face at a point that is interior to the edge-use.

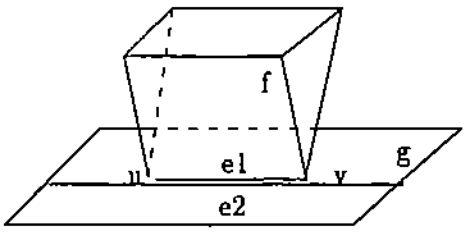

Figure 2.11 No intersection edge after neighborhood analysis

6. Vertex/vertex intersection: a vertex intersects a vertex of the other face.

Since two paired edge segments may not intersect, and in order to avoid unnecessary computation, we do not analyze the three dimensional neighborhood of an intersection unless it is an end point of the intersection of two paired edges or one isolated vertex and a paired edge or two isolated vertices. However, it may be the case that although two paired edge segments intersect each other on l, the intersection of the two segments does not belong to the intersection solid. This would be determined by the 3-D neighborhood analysis of the end points. In Figure 2.11, the intersection of paired e_1 and e_2 is e_1. But after neighborhood analysis at vertices u and v, e_1 is not an edge of the intersection solid. This applies equally to a vertex and paired edge intersection as well as to two intersecting isolated vertices.

When describing the three dimensional neighborhood analysis, in what follows, we assume that the intersection point belongs to the intersection of two paired edge or an isolated vertex and a paired edge or two isolated vertices. For each generic case, we outline the three dimensional neighborhood analysis as follows:

1. Edge/face intersection: An interior cross intersection must induce a new intersection edge. Two edge-uses of the new edge will be placed on f and g. Because the intersection point subdivides the edge, it should be put into the edge's subdiv. Using dot product, the proper traverse edge-uses are determined at the point on f and g.

2. Edge/edge intersection: Use the edge/edge intersection transferring algorithm to transfer edge/edge intersection to adjacent faces of both edges. The intersection point subdivides both edges and should be put into both edges' subdiv. Depending on the other end point of the intersection edge segment, this edge/edge intersection point may or may not induce a new intersection edge. If a new edge is created, then traverse edge-uses can be determined similarly to edge/face case.

3. Edge/vertex intersection: Use edge/vertex intersection transferring algorithm to transfer the intersection to entities adjacent to the edge and the vertex. The vertex subdivides the edge and should be put into the edge's subdiv. Use each plane adjacent to the edge to clip away edge-uses at the vertex that are above the plane. To clip an edge-use eu incident to a vertex v by a plane P, we substitute the coordinates of the other vertex of the eu into P and evaluate the value of the expression. If the value is positive, then eu is above P and hence is in the exterior of the other solid. eu is discarded. If the value is negative, then eu is below P and hence is in the interior of the other solid. eu is kept for further analysis. Otherwise, eu must be on P. Then we decide whether P is in the interior of a volume-enclosing pair centered at eu. If P splits a volume-enclosing pair at eu, then eu is kept for further analysis; or else it is discarded. The set of out-going edge-uses adjacent to the vertex that are left after clipping by all the faces incident to the edge should be stored in the intersection's travlist. They belong to the intersection solid and should be traversed later to assemble all the intersection shells. Note that for an edge-use eu in the travlist that is on a face incident to the edge, eu may be replaced by a new edge-use eu' that belongs to the intersection solid. This happens when a new edge created in later analysis which induces an edge-use eu' that is on the same face as eu is and has the same end points as eu.

4. Vertex/face intersection: Transfer intersection to edges adjacent to the vertex. Use the face to clip away edge-uses that are above the plane containing the face

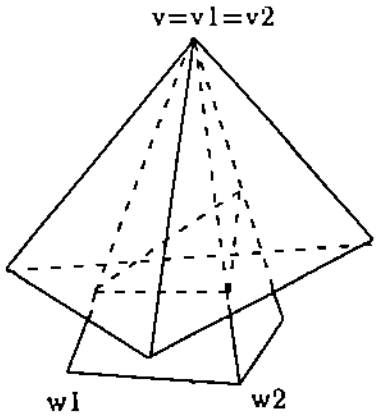

Figure 2.12 Vertex/vertex intersection neighborhood analysis

according to the procedure described in the edge/vertex case. The set of edge-uses that are below the plane will be put into the travlist of the intersection point.

5. Vertex/edge intersection: Same as the edge/vertex case.

6. Vertex/vertex intersection: We use planes adjacent to v_1 to clip each edge-use adjacent to v_2 and, vice versa, planes of v_2 to clip each edge-use adjacent to v_1 by the above described procedure. The union of the two sets of edge-uses left after clipping must be in the interior of both input solids and hence belongs to the intersection solid. In Figure 2.12, edge-uses vw_1 and vw_2 are the result of clipping and they both belong to the intersection solid.

We show next how the generic intersection analysis is applied to analyze intersection of two paired edges or an isolated vertex and a paired edge or two isolated vertices. Due to the large amount of spatial configurations, it is unnecessary and tedious to list every case of analysis in detail. Thus, we only give some examples. Other cases should be easy to derive in a similar way.

2.4.2.2 Intersecting Two Paired Edges

Let e_1, e_2 be the two paired edges and u_i, v_i are the starting and ending vertex of e_i, $i = 1, 2$, respectively. Then the nine possible intersection arrangements of are shown in Figure 2.13. The intersection of e_1 and e_2 is an edge formed by appropriately chosen end points of e_1 and e_2. Because u_i and v_i can either be a cross intersection point or a vertex intersection point, so combining intersection types of the end points of an intersection edge, there are $9 \times 4 = 36$ different cases when intersecting two paired edges.

Take Figure 2.13 (a) for example. The intersection of e_1 and e_2 is an edge segment with u_2 and v_1 as its end vertices. Now, there are four subcases according to the intersection types of u_2 and v_1: u_2 and v_1 are both cross intersections; u_2 is a cross intersection and v_1 is a vertex intersection; u_2 is a vertex intersection and v_1 is a cross intersection; u_2 and v_1 are both vertex intersections; see Figure 2.14.

We describe now the work involved in intersecting e_1 and e_2 in each cases.

(a) u_2 is a cross intersection. So u_2v_1 must be an edge of the intersection solid. We create a new edge u_2v_1 and two edge-uses of it to be placed on f and g. Suppose a is the edge-use of f that intersects g at v_1 and b is the edge-use of g that intersects f at u_2. Both u_2 and v_1 are subdividing intersections and are stored in the subdiv of their respective edges. The traverse edge-uses at u_2 and v_1 can be determined by applying dot product on a, b and normals of plane P and Q. At u_2: The traverse edge-use on f is from u_2 to v_1 and on g is along a's direction; At v_1: The traverse edge-use on f is along b's direction and on g is from v_1 to u_2. Note that at this time the traverse edge-uses at u_2 and v_1 is not complete. After all the faces incident to a have intersected f and all the faces incident to b have intersected g, then u_2 and v_1's traverse edge-uses will be completed.

(b) A new edge u_2v_1 and two edge-uses of it will be created. u_2 gets the same analysis as described in (a). For v_1, we first transfer intersection v_1 to all the

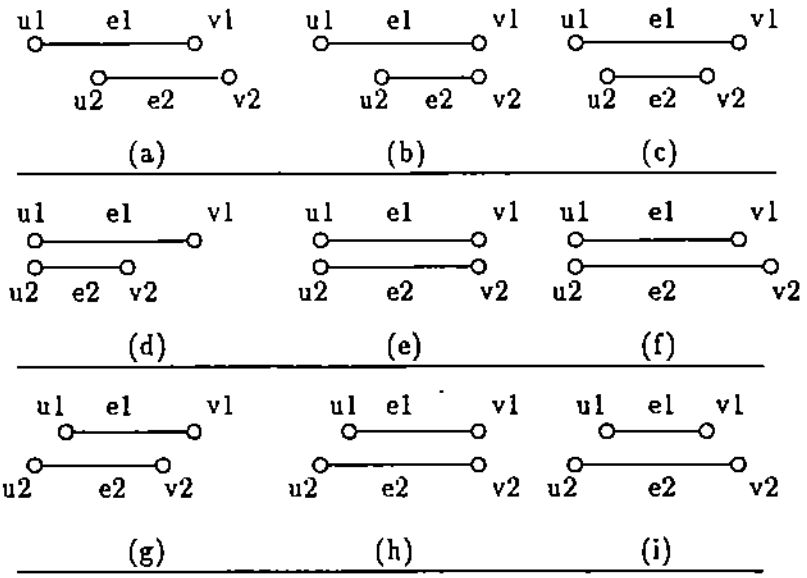

Figure 2.13 Intersecting two paired edges

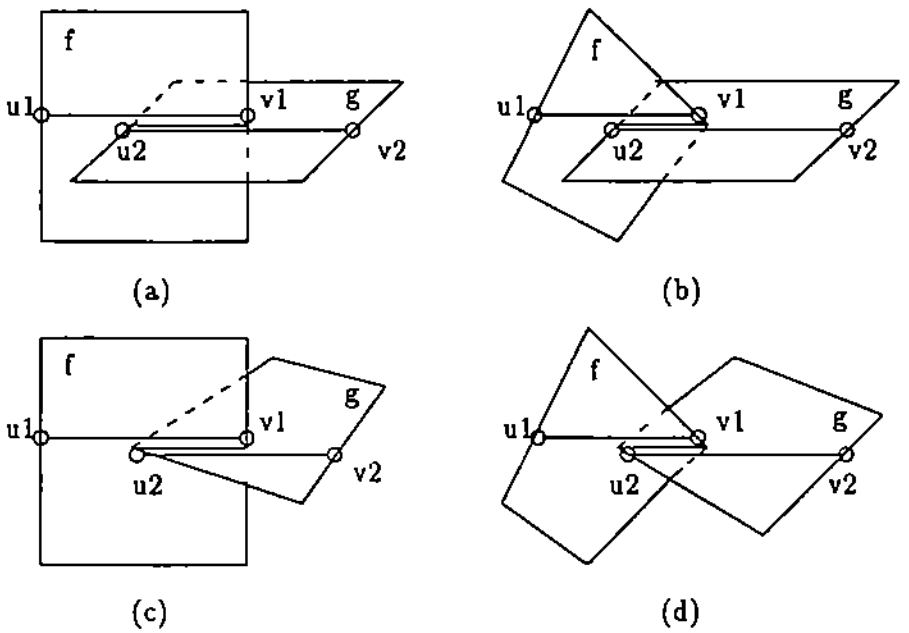

Figure 2.14 Example of intersecting two paired edges

edges adjacent v_1 according to the algorithm described earlier. Then use plane Q containing g to clip away edge-uses incident to v_1 that are above Q. Now using dot product properly, the traverse edge-uses are determined at u_2 and v_1 on f and g.

(c) Same as (b) with u_2 and v_1 reversed.

(d) Apply the analysis on v_1 in (b) to both u_2 and v_1.

2.4.2.3 Intersecting a Paired Edge with an Isolate Vertex

Assume now that e_2 is an isolated vertex. Since e_1 and e_2 intersect, there can be three cases: e_2 is in the interior of e_1; e_2 is equal to u_1 in coordinates; and e_2 is equal to v_1 in coordinates, see Figure 2.15. Furthermore, according to the intersection types of u_1 and v_1, there are $4 \times 3 = 12$ distinct cases in total.

Again, we take Figure 2.15 (a) for example. Now e_2 is in the interior of e_1. Depending upon the type of u_1 and v_1, we have the following cases:

1. u_2 and v_1 are both cross intersections: This is the case where e_2 intersects a face f in a point that is interior to f. We apply the generic intersection analysis of vertex/face to e_2 and f and create a new vertex of intersection if the clipped edge-uses in e_2 travlist is not empty.

2. u_2 is a cross intersection and v_1 is a vertex intersection or u_2 is a vertex intersection and v_1 is a cross intersection: Same as previous case.

3. u_2 and v_1 are both vertex intersections: Now e_1 must be an edge of an input solid. We have a case of edge/vertex intersection and apply the corresponding generic analysis accordingly to determine whether a new intersection vertex is created.

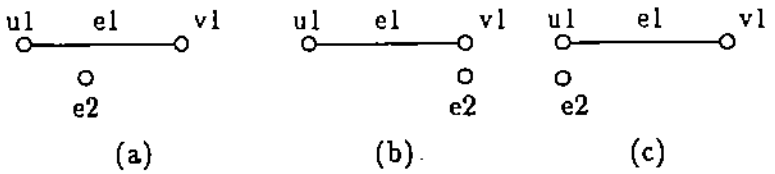

Figure 2.15 Intersecting a paired edge and an isolated vertex

2.4.2.4 Intersecting an Isolated Vertex with a Paired Edge

This can done in the same way as intersecting a paired edge with an isolated vertex.

2.4.2.5 Intersecting Two Isolated Vertices

Apply the analysis described in vertex/vertex generic intersection. If there are no edge-uses left after clipping, then both vertices are abandoned. Otherwise, a new vertex having the clipped set of edge-uses as its travlist is created and it belongs to the intersection solid.

2.4.3 Intersecting Two Coplanar Faces

Generically, for two coplanar intersecting faces, there are the following intersection configurations among vertices and edges.

1. Edge/Edge intersection: an edge-use intersects an edge-use of the other face at a point that is interior to both edge-uses.

2. Edge/vertex intersection: an edge-use intersects a vertex of the other face at a point that is interior to the edge-use.

3. Vertex/edge intersection: a vertex intersects an edge-use of the other face at a point that is interior to the edge-use.

4. Vertex/vertex intersection: a vertex intersects a vertex of the other face.

When intersecting two coplanar faces, we first intersect each pair of edges to get all the intersection points. The coordinates of an intersection point are now computed by intersecting two lines that contain the two respective edges. After an intersection point is computed, we apply proper intersection information transfer and 3-D neighborhood analysis at that intersection point. The neighborhood analysis is the same as the corresponding transversal intersection neighborhood analysis. Hence we do not elaborate further. With the set of out-going edge-uses at each intersection point determined and stored in its travlist, we next traverse each intersection point on the plane that contains both faces to assemble the intersection loops of the two faces. After traversing, if there are loops left that do not intersect any loops of the other face, we apply a 2-D containment tests on these loops to collect the ones that are in the interior of the other face. We use the algorithm that has been described in section 3.3.4 and hence omit the description.

2.4.4 Assembling Intersection Shells

After collecting all the intersection points, we traverse each intersection along its traverse edge-uses to assemble all the intersection shells. Assume that all the analyzed intersection points are collected in a list called *int-list*.

2.4.4.1 Traversing a Loop

We first describe how to traverse out a loop on a face.

1. Locate an intersection point that still needs to be traversed: Pick the first vertex v from *int-list* and delete v from *int-list*. If v's travlist is empty, we discard v and pick the next one in *int-list*. Repeat the process until either we find a vertex v that has nonempty travlist in which case we proceed to traverse the vertex; or *int-list* is empty in which case the traverse is done.

2. Start traversing v: The travlist of v is a list whose element is itself a list consisting of a face and a set of out-going edge-uses. Pick the first element from v's

travlist. Set the current traversing face f and the current out-going edge-use eu. Also we mark v as the starting vertex.

3. Move to the next vertex nv: Assume that e is the edge of eu. While moving, the adjacency information is incrementally built at v and nv. There are the following cases when determining nv along eu:

   - If e is a new edge, then nv is the other end vertex of eu. No new edge is created.
   - Otherwise, e is an edge of an input solid. If e has a nonempty subdiv, then sort the list of subdividing intersection vertices in subdiv if it was not marked as sorted. The nv can be decided from the value of subdiv and counting how eu is oriented with respect to e. There are two possibilities for nv: it is a subdividing vertex in e's subdiv or an end vertex of e. In either case a new edge connecting v to nv and corresponding edge-uses for faces adjacent to e will be created. Accordingly we replace corresponding out-going edge-uses at v by newly created edge-uses. If nv is a subdividing vertex, it must have been analyzed before. So the travlist of nv is complete. Therefore only appropriate replacement of out-going edge-uses in the travlist by new edge-uses is needed. If nv is an end vertex of e, we first make a copy of the end vertex and assign nv the copy of the vertex. Since nv is completely in the interior of an input solid, the out-going edge-uses at nv are precisely those out-going edge-uses at nv's adjacency. After establishing its travlist, we push nv into *int-list*. It will be further traversed later, on other adjacent faces.

4. Connect edge-uses and continue traversing: Edge-uses along traversing are connected using the next field of the edge-use structure. After nv located and analyzed if necessary, make nv current and continue traversing nv on the current face.

5. Report a Loop: When the current vertex is equal to the starting vertex, a loop of the current face has been found. Construct a new loop which points to the current face and has an index edge-use. Push the loop into the list of traversed loops. Go to step 1 to start traversing remaining loops.

After traversing, we have found loops of all the intersection shells. These loops and other loops when necessary form intersection faces. To create each individual intersection shell, we need to further collect assemble appropriate loops into faces, a process that will be described later.

Note that in the above algorithm, we do not tie our analysis to a particular traversing paradigm. Therefore, it should work for curved surfaces with intersection points properly computed and analyzed. Then the same algorithm will trace out loops of curved surfaces.

2.4.4.2 Gluing Loops at a Nonmanifold Vertex

When there is a nonmanifold vertex on a face, the traversing algorithm may trace out several loops that all have the vertex in common. Consider the left picture of Figure 2.16, two loops are traversed out for face f. However, there actually is only loop l on f since v is a nonmanifold vertex connecting the two loops l_1 and l_2. We need to reconnect the next field for edge-uses incident to the nonmanifold vertex to make multiple loops a single one as shown in the right picture of Figure 2.16.

To collapse multiple loops at a nonmanifold vertex, we first radially sort all the edge-uses incident to the vertex and then pair them into a list of nonarea-enclosing pairs. Now reconnect next of the first edge-use (coming-in) of a pair to the second edge-use (out-going) of the same pair.

2.4.4.3 Collecting Nonintersecting Loops of a Face

Even when two faces intersect transversally, there may still be loops left on a face which belong to the intersection solid. In Figure 2.17, assume that s_1 and s_2 both

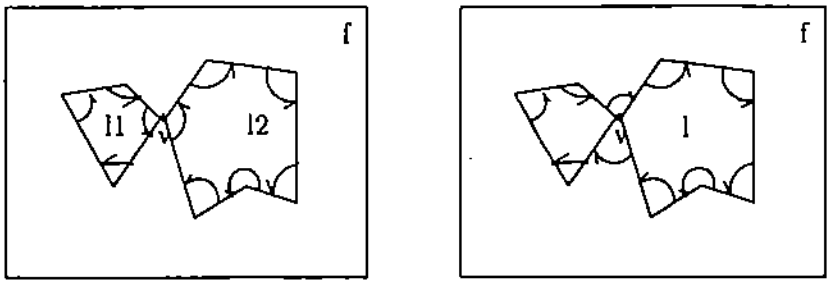

Figure 2.16 Glue loops at a nonmanifold vertex into a single loop

enclose infinite volumes. Each face of s_1 either does not intersect g or intersects g transversally. After intersection analysis, the traversing algorithm traverses out one intersection loop of g. But the original loop of g has not been put into the intersection solid since it does not intersect any face of s_1. Consequently, all the faces adjacent to edge-uses of the loop of g, which in fact are all parts of the intersection solid, are not traversed by the traversing algorithm.

Therefore, after intersection loops have been constructed for each face, we check whether there exists any loop on the face which did not intersect any face of the other solid. In case we find such a loop, we perform a two dimensional containment test on the loop and the constructed intersection loops. If the loop does belong to the intersection solid, then we include it in the list of loops of the intersection face and then apply the traversing algorithm to each vertex of the loop to traverse out all the adjacent faces that also belong to the intersection solid.

2.4.4.4 Assembling Faces of an Intersecting Shell

Having found loops for all the intersection faces, we are ready to assemble faces into intersection shells. This is basically a depth first search on all the intersection vertices including those belong to nonintersecting loops discoved by 2-D containment test. For Figure 2.17, after assembling, there is only one intersection shell.

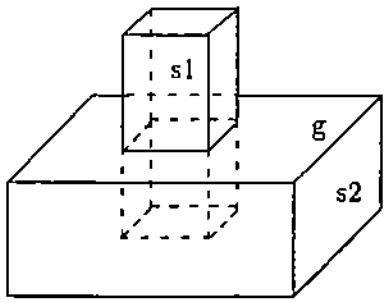

Figure 2.17 Collect nonintersecting loop into intersection solid

```
procedure GetShellFace(v)
begin
    mark v as 'old';
    for each face f adjacent to v do
        if f is new do
            mark f as 'old';
            push f into the face list of the shell;
        endif;
        for each vertex w of f do
            if w is new do
                GetShellFace(w);
            endif;
        endfor;
    endfor;
end
```

Algorithm: Collect faces of an intersecting shell

Input: A list V containing all the intersection vertices;

Return: A list of shells each of which is a list of faces bounding the shell;

while there exists a vertex v in V that is 'new' do

GetShellFace(v);

endwhile;

2.4.5 Shell Containment Test

Finally, we are ready to find those nonintersecting shells that belong to the intersection solid: They are shells of solid A that do not intersect any shells of solid B but are in the interior of B and, vice versa, shells of solid B that do not intersect any shells of A but are in the interior of A. We apply three dimensional containment tests on these shells with respect to those already discovered intersection shells. The three dimensional containment algorithm is essentially a line/solid classification algorithm. We again follow the 3-D containment test algorithm described in section 3.3.4 of the book by Hoffmann [12] and do not further describe it here.

2.5 Complement and Union Operations

If a solid has no vertices, edges and faces, then it is either an empty space or the whole universe. To complement such a solid, we negate the universe of the solid. Otherwise, to complement a solid, we perform the following:

1. Negate each shell's orientation infvol and each loop's orientation orient.

2. Multiply each plane equation by -1 and negate each face's orient indicating the inversion of the plane equation that contains the face.

3. Change the volume-enclosing pairs at each edge as follows: Move the first edge-use to the last and then pair every two consecutive edge-uses into a new volume-enclosing pair. Note that the volpair of each edge-use has to be modified accordingly.

Finally, the union of solid A and solid B by de Morgan's law is $\neg(\neg A \cap^* \neg B)$ where $\neg A$ is the complement of A.

2.6 Implementation

We implemented the above described algorithm as a solid modeler. The program was written in Common-Lisp first on a Symbolics 3650 lisp machine and has been also ported to a Sun Sparc Workstation.

3. DISCUSSIONS ON ROBUSTNESS ISSUES IN BOOLEAN ALGORITHMS

When floating point numbers are used to represent geometric data, inaccurate numerical results of geometric computation are inevitable. Failures of geometric algorithms, in consequence, occur usually when different numerical computations are used to determine the same symbolic fact, for example, incidence relations among different geometric entities. When performing Boolean operations on polyhedral solids, there are many incidence relations among vertices, edges and faces of the two input solids. Hence, there can be many chances for correct Boolean algorithms to fail on certain input in floating numbers.

We analyze in this chapter the spatial configurations of vertices, edges and faces between two input solids that could possibly result in failures of Boolean operations. We focus on analyzing incidence structures between the two input solids. We have observed the following facts.

First, the floating point arithmetic does not preserve the arithmetic laws of symmetry, associativity and transitivity. In consequence, the sequence of carrying out geometric computations using floating point arithmetic may affect the decisions about incidence relations among different geometric entities. Topological decisions are generally interrelated. To avoid inconsistencies, we have to coordinate the decision making and make sure that later decisions do not conflict with earlier decisions. Many geometric algorithms are found to be inadequate to making consistent topological decisions. They do not account for the dependencies among topological decisions and apply indiscriminately numerical procedures and the corresponding tests.

Second, incidence relations are usually derived differently in different algorithms. Therefore, the types of failure that could occur are algorithm dependent. There are heuristic approaches that have been proposed to make Boolean algorithms more

robust. In our algorithm, we try to make sure that each intersection is computed only once and each incidence relation is tested only once. This heuristic approach works well for many practical applications. But, there is no known provably robust Boolean algorithm that uses finite precision arithmetic.

From now on, we assume that solids satisfy a minimum feature separation condition when discussing robustness issues concerning Boolean intersection of two solids. Our minimum feature separation condition implies that with respect to a given tolerance ϵ, an edge of a solid cannot be shorter than ϵ, two vertices of a solid, if not connected by an edge, cannot be closer than ϵ and the angle between two faces adjacent to an edge cannot be smaller than a prescribed degree dependent on ϵ.

3.1 Intersection Derivation

When performing Boolean operations on solids, no new geometric datum is introduced if each solid is represented as the intersection of half spaces, where a half space is given by an oriented plane. However, the vertices of the new solid are represented only implicitly as intersections of a number of planes. We represent vertices of a solid explicitly and therefore, we need to compute vertices of the intersection solid. The vertices of the intersection solid can be classified into two classes: the class of vertices that belong to input solids and the class of new vertices that are derived from intersecting two solids.

A new vertex is derived when an edge l of the first solid intersects a face f of the second solid. The generic case shown left in Figure 3.1 is when l intersects f transversally. In the right of Figure 3.1 is the degenerate case when l is in the plane containing f. If l and f intersect transversally, the new vertex is derived unambiguously by intersecting the line segment and the plane containing f. Numerical errors, when computing the coordinates of the intersection, may perturb the point coordinates. But, they cannot cause inconsistencies in the adjacency structures of an intersection vertex. The problem may occur, however, when l is in the plane containing f. Right in picture of Figure 3.1, the intersection vertex may be derived

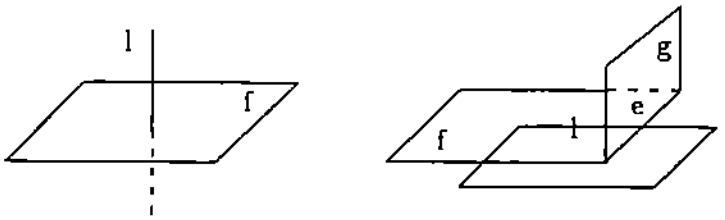

Figure 3.1 Intersection Derivation

differently by either intersecting l and face g or intersecting l and an edge e of f. If the algorithm allows both computations to coexist and the computed intersections do not agree under a floating point equality test, then we have inconsistent intersections.

To prevent this type of inconsistency, we transfer intersection information. For instance, after the face containing l is first intersected with face g, we will next intersect g and the line representing the intersection of the two planes that contain the two faces. During the process, we find out that l intersects e. In that case, we immediately perform edge/edge cross intersection transfer. Later, when l intersects e of face f, the intersection is retrieved from the appropriate field of the edge structure of l or e rather than recomputing a new intersection vertex.

When a vertex of an input solid is on an edge or a vertex of the other input solid, the vertex may or may not belong to the intersection solid depending on the 3-D neighborhood analysis. To avoid possible multiple decisions, edge/vertex or vertex/vertex intersection information transfer is equally valuable, see Figure 2.6.

3.2 Inconsistencies from Incidence Determination

Incidences between two geometric entities are usually determined by first computing some numerical quantities and then applying corresponding tests to the results.

The order of computations may lead to inconsistencies. We now analyze generic incidence structures to see which ones are sensitive to numerical errors and which ones are robust.

3.2.1 Face/Face Incidence

We can find, with a robust method, whether two faces are coplanar if we sacrifice efficiency. Specifically, the algorithm for intersecting two faces has to be structured in the following way: For each face f_1 of the first solid, we loop through every face f_2 of the second solid and test whether f_1 is coincident with f_2. Testing whether f_1 is coincident with f_2 is done by testing whether each vertex of f_1 is on the plane containing f_2. If at least three vertices of f_1 are found to be on the plane containing f_2, we declare that all the vertices of f_1 are on the plane containing f_2 and so f_1 and f_2 are deemed coplanar when both planes have the same orientation[14]. If such a coplanar face f_2 is discovered, we perform 2-D polygonal intersection of f_1 and f first and then proceed normally. If none of the faces of the second solid is coincident with f_1, we perform the usual procedure of intersecting f_1 and each face of the second solid consecutively. This procedure works most of times if minimum feature separation conditions are met by input solids. As outlined, the procedure is inefficient and hence is not used by any practical Boolean algorithms.

We found that two approaches that are commonly used in practice have flaws and may cause inconsistencies.

The first approach is the same as intersecting two transversal faces: Intersect each edge of the first face with the plane containing the second face. If all the edges of the first face lie on the plane containing the second face, then the two faces are coplanar. This approach is not robust. Take Figure 3.2 as the example. Assume that P is the plane containing face f_1. We may have the following scenario: When intersecting face g_1 and f_1, e_1 is found to be on plane P; next, when intersecting face g_i and f_1, e_i, $i = 2, 3, 4$, is found to be not on plane P; but, when intersecting face g_5 and f_1, e_5 is found to be on plane P. So, when finally intersecting f_1 and the face, call it f,

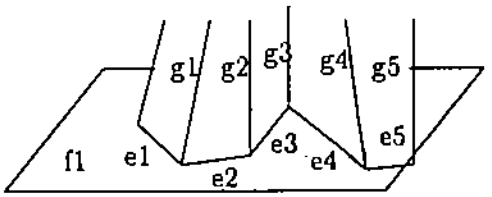

Figure 3.2 Face/face incidence inconsistency

containing e_is, $i = 1, \ldots, 5$, we have inconsistencies. Since e_1 and e_5 are on f_1, f and f_1 ought to be coplanar. But, since e_2, e_3 and e_4 are not on f_1, so f and f_1 cannot be coplanar. If we consider f_1 and f to be coplanar, we have to redo the analysis of intersecting f_1 and g_i, $i = 2, 3, 4$. Such backtracking is usually not carried out in intersection algorithms since it may involve too much work. The problem can be prevented if we intersect f_1 and f first in the manner described before. Note that if e_1 and e_5 happen to be consecutive on f, we may be able to avoid the inconsistencies by proclaiming that all the vertices of f are on plane P after discovering that e_1 and e_5 are on plane P. However, we generally cannot predict the best sequence in which to intersect faces. The basic philosophical problem we have is that we propagate only the consequences of equality, not of inequality. Hence we cannot eliminate inconsistencies associated with the two coplanar faces.

The second approach, adopted by our algorithm, is to test whether the two oriented planes containing both faces, f_1 and f_2 respectively, are the same before actually intersecting edges of f_1 with the plane containing f_2. If two faces are found to be coplanar, we declare all vertices of f_2 to be on the plane containing f_1 and then perform a 2-D polygon intersection. Note that deeming all vertices of f_2 to be on the plane containing f_1 is part of the intersection information transfer. Now when intersecting another face f with f_1, where f is adjacent to f_2 in the edge e, then we know that e is in the plane containing f_1. Thus, its vertex position are not reevaluated. This approach has similar inconsistency problems as the first approach. Suppose two

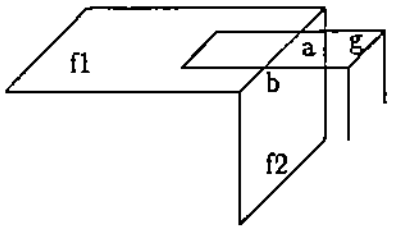

Figure 3.3 Face/face incidence inconsistency — another example

faces f_1 and f_2 are initially found not to be coplanar. But later, when intersecting f_1 with other faces, we may discover that at least two edges of f_2 are on the plane containing f_1. Hence, f_1 and f_2 should have been deemed coplanar in the first place.

Generally without exhaustive testing, the existence of two nearly coplanar faces is the single most important source of inconsistencies. No matter what kind of heuristic approach is applied, we cannot circumvent the problem.

There are other kinds of inconsistencies that appear to be different from the inconsistencies caused by face/face coincidence. However, we have found that many are actually rooted in the same face and face coplanarity inconsistency problem. For instance, another possible intersection inconsistency case is shown in Figure 3.3. When face g intersects face f_2, the two intersections a and b need not be on the plane containing the face f_1. For example, a and b are both below the plane containing f_1. Then after the g and f_2 intersection, an edge connecting a and b is created. Later when g and f_1 intersect, we find out that g and f_1 are coplanar. The two intersections created by intersecting g and f_1 may be different from a and b. And a redundant edge may also be created.

3.2.2 Edge/Face Incidence

Intersecting an edge and a face can be problematic when the edge is close to an edge of the face. In Figure 3.4, when l intersects face f_1, the intersection point a is in

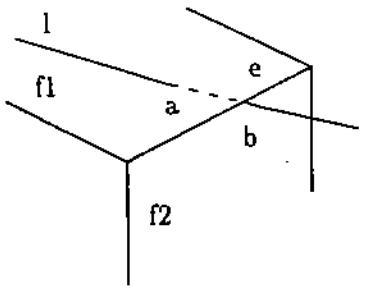

Figure 3.4 Edge/face incidence inconsistency

the interior of face f_1. However, when l intersects face f_2, it is found that l intersects an edge of e that is shared by f_1 and f_2. The inconsistency may be avoided if we intersect l and f_2 first and then perform edge/edge cross intersection information transfer. However, finding whether l intersects an edge of the other solid can only be done by exhaustively testing the intersection of l with all the edges of the other solid.

3.2.3 Vertex/Face Incidence

The inconsistency of vertex/face incidence is, in fact, induced by edge/face incidence inconsistency. In Figure 3.5, vertex v of l intersects face f_1 in the interior of f_1 and l is not in the plane containing f_1. However, when l intersects face f_2, it is found that l intersects an edge of e that is shared by f_1 and f_2.

3.2.4 Edge/Edge, Vertex/Edge and Vertex/Vertex Incidences

When any one of the edge/edge, vertex/edge and vertex/vertex incidences is discovered, the intersection information will be propagated to the adjacent structures of the edge or the vertex by intersection transfer algorithms. Hence, no inconsistencies will result from these three incidences.

- If an edge/edge intersection is found, the intersection vertex p must be in the interior of both edges and must not be too close to any end vertex of the two

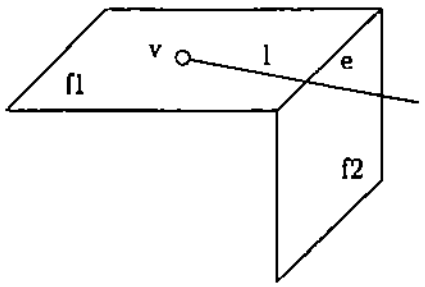

Figure 3.5 Vertex/face incidence inconsistency

edges by the minimum feature separation assumption. We immediately transfer intersection information to faces adjacent to both edges. When two faces, adjacent to the two edges respectively, intersect later, the same intersection vertex, p, will be returned faithfully.

- If a vertex v is found to be on an edge e in the edge interior (or an edge e passes through a vertex v), the intersection is transferred to edges and faces adjacent to v and faces adjacent to e. No inconsistencies can occur later.

- Finally, if vertex v_1 and vertex v_2 are discovered as vertex/vertex intersection, assume that p is the newly created intersection vertex, then all edges and faces adjacent to v_1 are known to intersect v_2 at p and vice versa, all edges and faces adjacent to v_2 are also known to intersect v_1 at p. So subsequent face and face, or edge and edge, intersection will return the same intersection vertex p.

Note that intersection information transfer plays an key role here to avoid inconsistencies that might be caused by incidences.

3.2.5 Summary

To summarize, we note that there are basically two types of incidence, face/face incidence and face/edge incidence, that may cause inconsistencies. Applying some

sophisticated heuristics, for example, the incidence test of Hoffmann, Hopcroft and Karasick [14], one may be able to avoid the type of inconsistencies caused by face/edge incidence. Such heuristics usually entails only little more work and is still practical. However, face/face incidence is more difficult to deal with robustly. We consider edge/edge, edge/vertex (vertex/edge) and vertex/vertex incidence robust if proper intersection information is transferred.

3.3 Inconsistencies from Neighborhood Analysis

When an incidence of two geometric entities is found by numerical computations in floating point arithmetic, it only means that we cannot distinguish them with the arithmetic precision used. The two geometric entities may, in fact, not coincide but are deemed as coincident. When this happens, it will affect 3-D neighborhood analysis at intersection points and can cause inconsistencies in intersection neighborhood structures. The problem is that after deeming two geometric entities coincident, we have implicitly altered some geometric data associated the two geometric entities. For instance, if an edge e is claimed to lie in the plane P, then the dot product of the vector connecting end points of e and the normal of P should be deemed as zero also. However, this condition may not be true if the e/P coincidence is discovered by testing end points of e with P. Later, if the above dot product operation is used in analyzing 3-D neighborhood at e, we may construct inconsistent neighboring structures at e.

This kind of data dependence due to coincidence of two geometric entities causes problems when dependent data are used in performing 3-D neighborhood analysis at intersection points. The types of data that are affected can be plane normals of faces adjacent to a vertex v when v is found to be on an edge or in a face, or face direction vectors of edge-uses that are associated with an edge e when e is found to be passing through a vertex or intersecting another edge, etc.

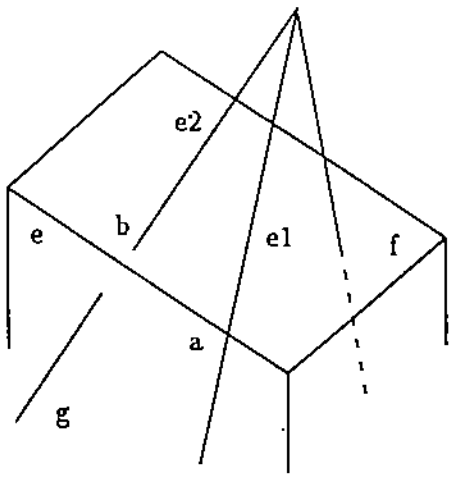

Figure 3.6 Edge/edge neighborhood analysis inconsistency

3.3.1 Vertex/Face, Edge/Face Neighborhood Analysis

If a vertex v is found to be in a plane P containing face f, inconsistencies in the neighborhood analysis at v happen when there are faces adjacent to v that are coincident to f. This is essentially the problem of face/face coincidence determination. Since edge/face neighborhood analysis reduces to vertex/face analysis of its end points, it has the same problems as vertex/face neighborhood analysis.

3.3.2 Edge/Edge Neighborhood Analysis

When analyzing an edge/edge neighborhood, inconsistencies caused by edge/edge incidence can be demonstrated as follows. In Figure 3.6, suppose edge e_1 of face g intersects edge e of f exactly at point a and edge e_2 of g and e of f are deemed as intersecting at point b by floating point arithmetic. Since a is an edge/edge transversal intersection, we first test whether e is in the interior of any wedge that is spanned at edge e_1. This is done by projecting all the face direction vectors of each volume-enclosing pair at e_1 and edge e itself into a plane which is perpendicular to edge e_1 and then determining whether the projected edge e is inside any projected area-enclosing

pair. Conceptually, because e now intersects both e_1 and e_2, it should lie in face g so that the projected e will coincide with the projected face direction vector of e_1 on g. But, our algorithm uses the original e for projection. Therefore, it claims that e is not in the interior of the wedge at e_1. After knowing the wedge at e and wedge at e_1 are both on the same side of the plane determined by e_1 and e, we collect all the faces, g among them, at e_1 into the neighborhood of a. Since g is an extraneous face, which does not have out-going edge-uses, at a's travlist, so later, the traversing algorithm will fail to trace out g starting from a.

The example shows the problem of data dependency induced by incidences. To circumvent the problem, we need to store the information that e is in face g and exploit derived incidence information when analyzing 3-D neighborhood. Although intersection transfer avoids multiple computations of an intersection point, it does not, however, solve such kind of data dependency problems.

3.3.3 Vertex/Edge or Edge/Vertex Neighborhood Analysis

Similar kind of inconsistencies, shown in Figure 3.7, can happen when analyzing 3-D neighborhood at vertex/edge or edge/vertex intersection. In the picture, e_1 intersects e at its vertex a and e_2 and e are deemed as intersecting at point b. After neighborhood analysis, point a gets an extra face g.

3.3.4 Vertex/Vertex Neighborhood Analysis

If e_1 and e intersect at one of their end points, shown in Figure 3.8, the same problem happens at the intersection point a.

3.3.5 Global Data Dependency

It should be noted that the data dependencies induced by incidence structures do not have to be local. Even if we take measures to avoid local inconsistencies, we may not be able to prevent global inconsistencies. Take Figure 3.9 for example. Assume that e_1 intersects e exactly at a, e_2 intersects e by floating point arithmetic at b and

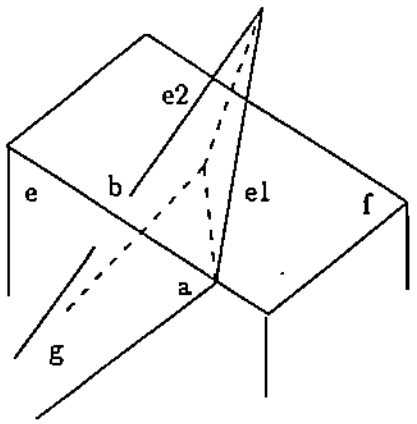

Figure 3.7 Vertex/edge or edge/vertex neighborhood analysis inconsistency

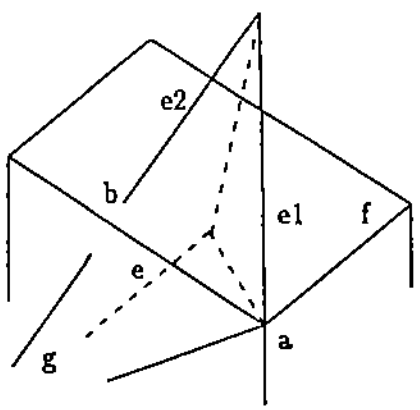

Figure 3.8 Vertex/vertex neighborhood analysis inconsistency

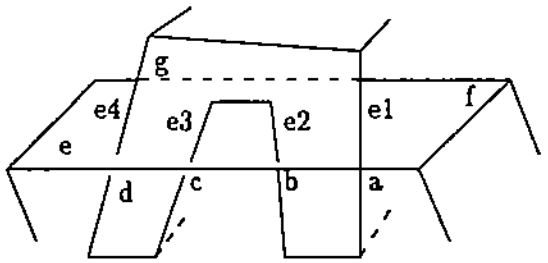

Figure 3.9 Global data dependencies

e_3, e_4 do not intersect e but intersect face f interior at c, d respectively. No matter how we place e, we cannot avoid inconsistencies. Since e intersects e_1 and e_2, it should be in the face g. However, e does not intersect e_3 and e_4, so it ought to be outside of face g.

One solution is to subdivide e into two segments and place two segments differently with respect to face g. But the vertices of the two segments belonging to the same face of the output solid are in fact not coplanar.

3.3.6 Summary

When an incidence of two geometric entities is found by using floating point arithmetic, it may invalidate certain geometric data of solids both locally and globally. In consequence, 3-D neighborhood analysis of intersection points based on the original geometric data is error prone. Systematic data perturbations to accommodate incidence decisions is found to be hard since incidences induce data dependencies that have global effects. Currently, most Boolean algorithms, if not all of them, base their neighborhood analysis on the original input data and therefore are not robust. In our experience, the majority of failures of our Boolean algorithm occur when the neighborhood analysis is incompatible with incidence determinations. The numerical

treatment of small features induced by deemed incidences in the neighborhood analysis is problematic. In the rational case, small features are always handled correctly, so that these types of failures are indeed numerical rather than algorithmic rooted.

4. POINT/LINE AND POINT/PLANE CLASSIFICATION

4.1 Introduction

The determination of on which side of a line or plane a point lies is called the problems of point/line and point/plane classifications. This classification is a fundamental geometric operation. It has been shown by Sugihara and Iri [33] that all computations deciding topological structure in the course of set-theoretic operations can be reduced to point/line or point/plane classification.

By determining the sign of the expression after substituting point coordinates into the line or plane equation, one can classify the point/line or point/plane relationship. However, with arithmetic typically performed in floating point numbers, numerical round-off and cancellation errors cause ambiguity when the point is close to the line or plane. A common heuristic is to choose a tolerance and to regard a point incident to a line or plane whenever it is within the chosen tolerance of the line or plane. This approach will not in general resolve incidence ambiguities satisfactorily, because the judgement of whether the point is within the tolerance region depends on the details of the computation. Equivalent geometric operations can lead to inconsistent judgements of the topological structure and thereby can cause program failures [12].

If we are able to classify point/line or point/plane incidence exactly, then we can determine the topological structure of geometric objects in geometric computations correctly. But, it is clear that an exact classification is not possible if the computation is carried out within the same arithmetic precision as the given input data. Sugihara and Iri [33] estimated a sufficient precision for the classification problem assuming that line or plane equations are primary data and each point is defined as the intersection of two lines in 2-D or of n planes in n-D. We consider the problem to determine the minimum distance between a point so defined and a line or a plane, and so

establish a lower bound on the precision needed for the classification problem. Since our necessary bound matches Sugihara's sufficient bound, it is optimal.

A related quantity, called "minimum feature" by some authors, is the minimum edge length. An edge is determined by two distinct points on a line. Thus a minimum edge is one in which the distance between the end points is minimum and the points have been defined as line intersections. We will show that the length of a minimum edge has the same order of magnitude as the minimum point/line or point/plane distance.

4.2 Background and Notations

Let $\mathbf{E}^n$ be the n dimensional Euclidean space and L be a positive integer. A plane in $\mathbf{E}^n$ (line when $n = 2$) is represented implicitly as

$$P: \ a_1x_1 + a_2x_2 + \cdots + a_nx_n + a_{n+1} = 0$$

We consider the set of planes (lines when $n = 2$)

$$S_L = \{ \ P \mid -L \leq a_i \leq L, \text{ for } 1 \leq i \leq n \text{ and } -(n-1)L^2 \leq a_{n+1} \leq (n-1)L^2 \ \}$$

where the a_i are integers. Suppose each polyhedral object in $\mathbf{E}^n$ is represented by a list of planes from S_L. An intersection point in $\mathbf{E}^n$ is defined as the intersection of at least n distinct planes from S_L. Note that there are only finitely many representable objects and finitely many intersection points.

This approach of representing objects was first introduced by Sugihara and Iri in [33]. The seemingly peculiar choice for the constant term a_{n+1} provides a uniform distribution of planes or lines in S_L in the square region $-L \leq a_i \leq L$, $i = 1, \ldots, n$, as explained in [33] or section 4.3.1 of [12]. Figure 4.1 shows S_L when $n = 2$ in the region $-3 \leq x_1 \leq 3$ and $-3 \leq x_2 \leq 3$ with $L = 3$.

We consider the minimum point/line, point/plane distances and minimum edge length problems in 2-D and 3-D in the following sections. We assume L is a reasonably large positive integer. For example, L has at least two decimal digits. Our result shows

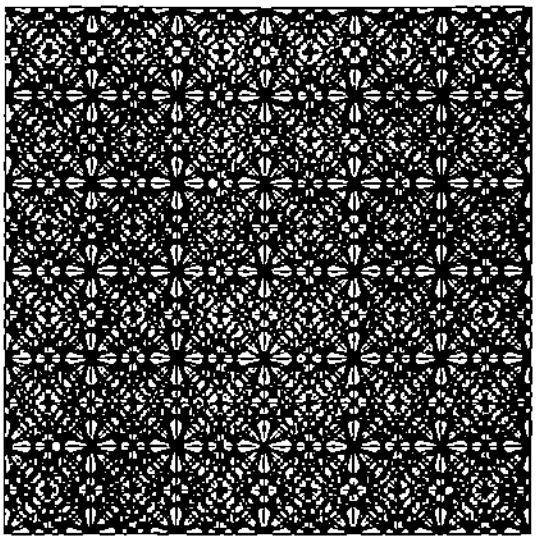

Figure 4.1 Representable lines in $|x|, |y| \leq 3$ when $|a_1|, |a_2| \leq 3$ and $|a_3| \leq 18$

that the minimum point/line distance and minimum edge length in 2-D are $O(L^{-3})$ and the minimum point/plane distance and minimum edge length in 3-D are $O(L^{-4})$. For the general n-D problem with $n > 3$, we have established a compact formula that gives bounds on the quantities.

4.3 2-D Minimum Point/Line Distance and Minimum Edge

Let $l_i : a_i x + b_i y + c_i = 0$, $i = 1, 2, 3$, be three lines. Then the distance of the intersection of l_1 and l_2 from the line l_3 is

$$\left| \frac{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}{\sqrt{a_3^2 + b_3^2}\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} \right| \tag{4.1}$$

We call (4.1) the point to line distance. Likewise, the distance between the intersections of l_1, l_2 and of l_1, l_3 is found to be

$$\left| \frac{\sqrt{a_1^2 + b_1^2} \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \left\| \begin{vmatrix} a_1 & b_1 \\ a_3 & b_3 \end{vmatrix} \right.} \right| \tag{4.2}$$

after some algebraic manipulation. The minimum of (4.2) clearly defines the minimum edge length, or minimum feature of polygons that can be represented by lines from S_L.

In the following, a quadrant includes the corresponding parts of the x and y-axes. If P_1, P_2 and P_3 are three points, then P_1-P_2-P_3 denotes that they are collinear and $OP_1 \perp OP_2$ means that OP_1 is perpendicular to OP_2.

We prove two theorems, one about the minimum distance of a point from a line and the other about minimum edge length. The proof of both theorems uses the following geometric interpretation: Consider (a_i, b_i) as a point P_i in a 2-D rectangular lattice of $[-L, L] \times [-L, L]$. Then the 2×2 determinant $\begin{vmatrix} a_i & b_i \\ a_j & b_j \end{vmatrix}$ is twice the signed area of the triangle (O, P_i, P_j), where O is the origin, and $\sqrt{a_i^2 + b_i^2}$ is the length of the vector from the origin to P_i.

In both proofs, we will maximize the denominator while keeping the numerator minimum. Since c_i appears only in the 3×3 determinant of the numerator, c_i can be chosen as 1 without affecting both minima. Therefore, the precision bound on the c_i does not influence our results.

Table 4.1 Descending attainable square root of $a_3^2 + b_3^2$

a_3	b_3	$\sqrt{a_3^2 + b_3^2}$
$\pm L$	$\pm L$	$\sqrt{L^2 + L^2}$
$\pm L$	$\pm(L-1)$	$\sqrt{L^2 + (L-1)^2}$
$\pm L$	$\pm(L-2)$	$\sqrt{L^2 + (L-2)^2}$
$\pm(L-1)$	$\pm(L-1)$	$\sqrt{(L-1)^2 + (L-1)^2}$
...	...	.....

4.3.1 The Minimum Point to Line Distance

Theorem 4.1 The minimum point to line distance, that is, the minimum of (4.1), is

$$\frac{1}{\sqrt{2}L(2L^2 - 2L + 1)}$$

Proof: Let $P_i = (a_i, b_i)$, $i = 1, 2, 3$, be three lattice points corresponding to the line coefficients. Since a_i, b_i and c_i are integers, the absolute value of the numerator is at least 1 (the case where the numerator is zero is uninteresting). Our first observation is that a numerator of 1 is necessary for (4.1) to be minimum. Suppose the numerator is 2. Since the absolute value of the denominator of (4.1) cannot be larger than $\sqrt{2}L \times 2L^2$, the resulting minimum value in this case is $\frac{1}{\sqrt{2}L^3}$ which is larger than the minimum stated in the theorem.

By enumerating possible values for each component of the denominator in a descending order, we would be able to find out maximum value of the denominator. Because of the symmetry between a_3 and b_3, we assume $|a_3| \geq |b_3|$. Table 4.1 shows the attainable values of $\sqrt{a_3^2 + b_3^2}$ in descending order.

Before constructing a similar list for the 2×2 determinant, we need to distinguish three different cases:

- P_1 and P_2 are in quadrants that share half of an axis;

- P_1 and P_2 are in quadrants that only share the origin;
- P_1 and P_2 are in the same quadrant.

For the first case, without loss of generality, let us assume P_1 to be in quadrant I and P_2 in quadrant IV. Then, the list is in the Table 4.2.

Now the largest possible denominator is $\sqrt{2L^2}2L^2$. But, in this case, there will be at least one column of the 3×3 determinant of the numerator that has a common factor of L so the numerator is at least L. Consequently, the value of (4.1) is not minimum.

Hence, excluding combinations of P_1, P_2 and P_3 that will result in a common factor of L or $L-1$ in either column of the 3×3 determinant of the numerator, the candidate values for the denominator are selected in the following way: For each value from Table 4.1 in descending order, find the first value from Table 4.2 such that the combination of P_1, P_2 and P_3 does not have a common factor L or $L-1$ in both the first and the second coordinate. In case there are combinations that derive the same value for the denominator, then all these combinations are considered. The result is collected in the Table 4.3.

Since Table 4.3 contains the largest values from both descending lists, it must also contain the maximum denominator for (4.1). Comparing the four numbers in the Table, the largest one is

$$\sqrt{2}L(2L^2-2L+1)$$

Now let $P_3=(L,-L)$, $P_1=(L-1,L)$, $P_2=(L,-(L-1))$ and $c_1=c_2=c_3=1$. Then,

$$\begin{vmatrix} L & -L & 1 \\ L-1 & L & 1 \\ L & -(L-1) & 1 \end{vmatrix} = -1$$

or the absolute value is 1 for the numerator. Thus the theorem holds for the first case.

For the second and third case, the following can be proved:

Table 4.2 Descending attainable 2×2 determinant

$P_1(a_1, b_1)$	$P_2(a_2, b_2)$	$\lvert det(P_1, P_2)\rvert$
(L, L)	$(L, -L)$	$2L^2$
(L, L)	$(L, -(L-1))$	$2L^2 - L$
(L, L)	$(L-1, -L)$	$2L^2 - L$
$(L-1, L)$	$(L, -L)$	$2L^2 - L$
$(L, L-1)$	$(L, -L)$	$2L^2 - L$
$(L-1, L)$	$(L, -(L-1))$	$2L^2 - 2L + 1$
$(L, L-1)$	$(L-1, -L)$	$2L^2 - 2L + 1$
...	...	.....

Table 4.3 Candidate denominator values

a_3	b_3	$P_1(a_1, b_1)$	$P_2(a_2, b_2)$	denominator
$\pm L$	$\pm L$	$(L-1, L)$	$(L, -(L-1))$	$\sqrt{L^2 + L^2}(2L^2 - 2L + 1)$
$\pm L$	$\pm L$	$(L, L-1)$	$(L-1, -L)$	$\sqrt{L^2 + L^2}(2L^2 - 2L + 1)$
$\pm L$	$\pm(L-1)$	(L, L)	$(L-1, -L)$	$\sqrt{L^2 + (L-1)^2}(2L^2 - L)$
$\pm L$	$\pm(L-1)$	$(L-1, L)$	$(L, -L)$	$\sqrt{L^2 + (L-1)^2}(2L^2 - L)$
$\pm L$	$\pm(L-2)$	(L, L)	$(L-1, -L)$	$\sqrt{L^2 + (L-2)^2}(2L^2 - L)$
$\pm L$	$\pm(L-2)$	$(L-1, L)$	$(L, -L)$	$\sqrt{L^2 + (L-2)^2}(2L^2 - L)$
$\pm(L-1)$	$\pm(L-1)$	(L, L)	$(L, -L)$	$\sqrt{(L-1)^2 + (L-1)^2}2L^2$
...	...	.....	.....	.....

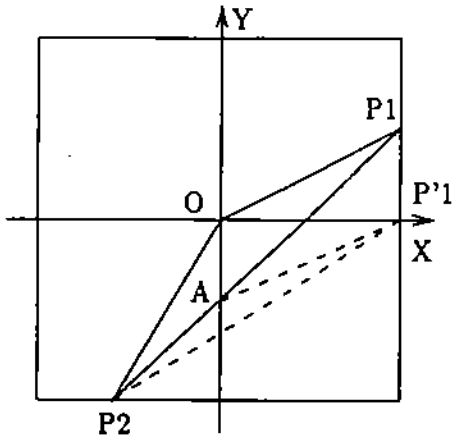

Figure 4.2 P_1 on $x = L$ and P_2 on $y = -L$

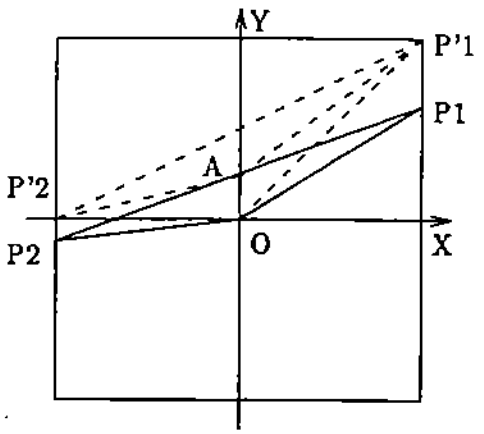

Figure 4.3 P_1 on $x = L$ and P_2 on $x = -L$

Claim The 2×2 determinant $\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}$ is less than or equal to L^2.

In fact, the claim is true even for non-lattice points P_1 or P_2.

To begin, let P_1 and P_2 be in the diagonal quadrants. Without loss of generality, we assume that P_1 is in quadrant I and P_2 in quadrant III. It is sufficient to prove that the area of the triangle formed by the origin O, P_1 and P_2 is less than or equal to $\frac{L^2}{2}$. Suppose either P_1 or P_2 is in the quadrant interior so that the absolute values of both the x and y coordinates are less than L, then one can extend line OP_1 or OP_2 to intersect the boundary lines $x = \pm L$ or $y = \pm L$ and choose the intersection point as P_1' or P_2'. The new triangle so formed $OP_1'P_2'$ has a larger area than the original triangle. Hence, without loss of generality, we assume P_1 and P_2 are on the boundary. By symmetry, there are only two distinct cases:

- P_1 is on $x = L$ and P_2 on $y = -L$;
- P_1 is on $x = L$ and P_2 on $x = -L$.

Denote the intersection point of the line P_1P_2 and the y-axis by A and let $\triangle OP_1P_2$ be the area of triangle OP_1P_2. Then, in the first case, choose $P_1' = (L, 0)$. So,

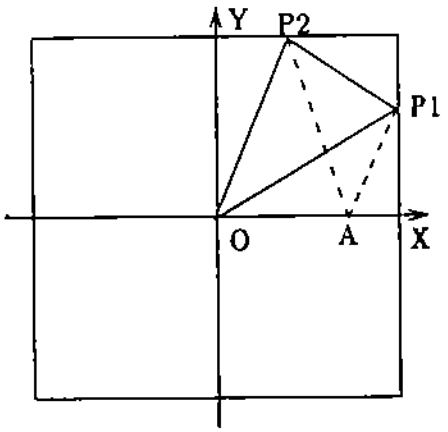

Figure 4.4 P_1 on $x = L$ and P_2 on $y = L$

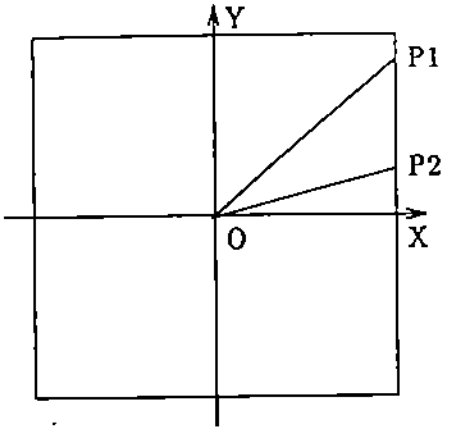

Figure 4.5 P_1 on $x = L$ and P_2 on $x = L$

$\triangle OP_1A = \triangle OP_1'A$, and

$$\triangle OP_1P_2 = \triangle OP_1A + \triangle OAP_2$$

$$= \triangle OP_1'A + \triangle OAP_2 \leq \triangle OP_1'P_2 = \frac{L^2}{2}$$

If P_1', A and P_2 are collinear (see Figure 4.2), then equality holds. In the second case, if P_1, O and P_2 are not collinear, then, depending on whether the y coordinate of A is positive or negative we can choose new points as follows:

- If the y coordinate of A is positive, choose $P_1' = (L, L)$ and $P_2' = (-L, 0)$ (see Figure 4.3). Then

$$\triangle OP_2P_1 = \triangle OP_2A + \triangle OAP_1$$

$$= \triangle OP_2'A + \triangle OAP_1' \leq \triangle OP_2'P_1' = \frac{L^2}{2}$$

  Again, we have equality when P_1',A and P_2' are collinear.

- If the y coordinate of A is negative, choose $P_1'(L, 0)$ and $P_2'(-L, -L)$. An argument similar to the above case shows $\triangle OP_1P_2 \leq \frac{L^2}{2}$.

Now, for the case of P_1 and P_2 are in the same quadrant, say quadrant I. Again, we have two sub-cases: *a*) P_1 is on $x = L$ and P_2 is on $y = L$; *b*) P_1 is on $x = L$ and P_2 is on $x = L$; see Figure 4.4 and Figure 4.5.

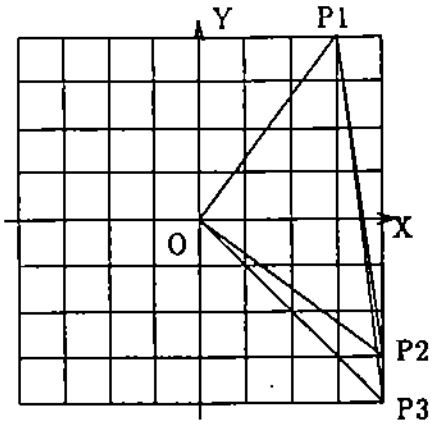

Figure 4.6 Arrangement of P_1, P_2 and P_3 for minimum point/line distance

For case a), let P_1A be parallel to line OP_2 and A be the intersection of P_1A and the x-axis.

$$\triangle OP_2P_1 = \triangle OP_2A = \frac{1}{2}|OA|L \leq \frac{1}{2}L^2$$

If b) is the case, then

$$\triangle OP_1P_2 = \frac{1}{2}|P_2P_1|L \leq \frac{1}{2}L^2$$

Therefore, the claim has been proved.

Finally, since $L^2 \leq L^2 + (L-1)^2 = 2L^2 - 2L + 1$, the previously chosen points P_1, P_2 and P_3 from the first case as well as suitably chosen c_i produce a numerator of 1 and so minimize (4.1). Therefore, the theorem holds □

Remark 1. In fact, one can prove theorem 4.1 by using geometric intuition. If we denote the 3×3 determinant of the numerator as $det(a, b, c)$, then (4.1) can be written as

$$\frac{|det(a, b, c)|}{|OP_3||OP_1||OP_2||\sin \angle P_1OP_2|} \tag{4.3}$$

Without the $|\sin \angle P_1OP_2|$ term, the minimum of

$$\frac{|det(a, b, c)|}{|OP_3||OP_1||OP_2|}$$

is obtained by making $|OP_i|$ as large as possible while at the same time keeping $|det(a,b,c)|$ as close to 1 as possible. Clearly, the largest denominator is to have all three P_is be $(\pm L, \pm L)$. But this would result in a common factor L in $det(a,b,c)$. If two of the three P_is are $(\pm L, \pm L)$, then, in order to avoid common factor in either the first or the second column of $det(a,b,c)$, the largest possible coordinates for the third one is $(\pm(L-1), \pm(L-1))$. But then

$$\sqrt{2}L\,\sqrt{2}L\,\sqrt{2}(L-1) < \sqrt{2}L(\sqrt{L^2+(L-1)^2})^2$$

Thus, the largest possible denominator is to have one $|OP_i|$ to be the largest value and the other two the second largest value from Table 4.1. By forming a shape shown in Figure 4.6 and choosing $c_i = 1$, three such points make $|det(a,b,c)| = 1$ and therefore the maximum denominator is $\sqrt{2}L(\sqrt{L^2+(L-1)^2})^2$.

Now, for the minimum of (4.3), the matter is simply to decide which points should be P_1 and P_2 so that $|\sin \angle P_1OP_2|$ will be as large as possible. Hence, the choice of P_1 and P_2 is to make OP_1 and OP_2 as close to perpendicular as possible. It happens that two points in Figure 4.6 expand 90° at the origin. So, these two points are chosen as P_1 and P_2 and the remaining one as P_3, see Figure 4.6. The arrangement of P_is obtained in this way matches precisely the arrangement of the algebraic proof.

Remark 2. From the proof, it is clear that the magnitude of the constant term c of the line equation does not affect the precision required for classification. Its magnitude does, however, affect the density of the lines in a specified grid region.

4.3.2 Minimum Edge Length

Theorem 4.2 The minimum edge defined by S_L has the length

$$\frac{1}{(2L^2-L)\sqrt{2L^2-2L+1}}$$

This length minimizes (4.2).

Proof: By our geometric interpretation, (4.2) can be rewritten as

$$\left| \frac{|OP_1| \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}{|OP_1||OP_3| \sin \angle P_1OP_3 \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} \right| = \left| \frac{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}{\sqrt{a_3^2 + b_3^2} \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} \right| \frac{1}{|\sin \angle P_1OP_3|} \tag{4.4}$$

The denominator of (4.4) is simply the product of the denominator of (4.1) and $\sin \angle P_1OP_3$. Because $|\sin \angle P_1OP_3| \leq 1$ and Table 4.3 is in the order of descending denominator of (4.4), if there is a row in the Table 4.3 that makes $|\sin \angle P_1OP_3| = 1$, then the maximum of the denominator of (4.4) must be among those rows in the Table 4.3 that have larger value than that of the row having $|\sin \angle P_1OP_3| = 1$.

Going through Table 4.3, both the fourth and the seventh row have $|\sin \angle P_1OP_3| = 1$. But, the denominator of the seventh row is less than that of the fourth row. Now, it is easy to check that the denominator of the fourth row is only less than the first and the second row in the table. By substituting the values of P_1, P_2 and P_3 from these three rows into the denominator of (4.4), we find that the fourth row has the largest value.

Therefore, if we chose $P_1 = (L-1, L)$, $P_2 = (L, -L)$ and $P_3 = (L, -(L-1))$ or $P_3 = (-L, L-1)$ (because of symmetry between a_3 and b_3, see Figure 4.7), then the maximum value of the denominator of (4.4) is $(2L^2 - L)\sqrt{L^2 + (L-1)^2}$. Finally let $c_1 = c_2 = 1$ and $c_3 = 1$ if $P_3 = (L, -(L-1))$ or $c_3 = -1$ if $P_3 = (-L, L-1)$. Then the absolute value of the 3×3 determinant of the numerator is 1.

Hence, the correctness of the theorem is established □

It can be easily checked that the configuration of P_1, P_2 and P_3 deriving a minimum edge has the same shape as the configuration deriving a minimum point to line distance. The explanation of this is similar to the remark after Theorem 4.1.

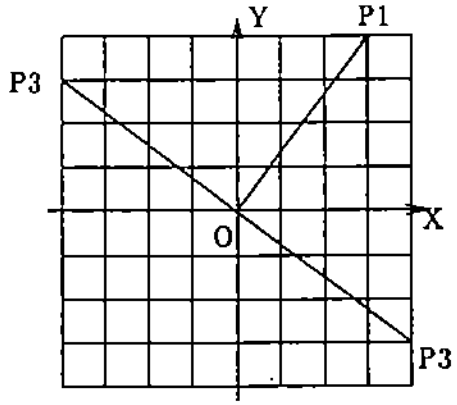

Figure 4.7 Arrangement of P_1 and P_3

Geometrically, (4.2) can be written as

$$\frac{|det(a,b,c)|}{|OP_3||OP_1||OP_2||\sin \angle P_1OP_3||\sin \angle P_1OP_2|}$$

The configuration of P_i without the sin terms in the denominator is known. So, to make (4.2) minimum, we should choose P_1 such that $\angle P_1OP_2$ and $\angle P_1OP_3$ are as close to 90° as possible. This requires P_1 to be in the position shown in Figure 4.6. With such choice, one of the angles of $\angle P_1OP_2$ and $\angle P_1OP_3$ is 90° and the other is slightly larger than 90°.

4.3.3 The Structure of the Two Minimums

By the proof of Theorem 4.2, the two 3-line arrangements that minimize (4.2) are as shown in the table

l_1	l_2	l_3
$(L-1)x+Ly+1=0$	$Lx-Ly+1=0$	$Lx-(L-1)y+1=0$
$(L-1)x+Ly+1=0$	$Lx-Ly+1=0$	$-Lx+(L-1)y-1=0$

For each (l_1, l_2, l_3) triple, it is easy to see that l_1 is perpendicular to either l_2 or l_3 (P_2 and P_3 are symmetric). This is not a coincidence. Although the above

two minimum edge line configurations are derived under the restrictions that P_1 is in quadrant I and $|a_3| \geq |b_3|$, we could have derived other minimum edge line configurations had we removed one or both of these restrictions. All other minimum line configurations are equivalent, but do not have P_1 in quadrant I or $|a_3| \geq |b_3|$. In the minimum edge line configurations, l_1 must be perpendicular to either l_2 or l_3, for, as explained before, either $\angle P_1OP_2 = 90°$ or $\angle P_1OP_3 = 90°$ in a minimum edge configuration. Hence, we obtain

Corollary 4.3 If l_1, l_2 and l_3 belong to S_L and the intersections of l_1, l_2 and l_1, l_3 form a minimum edge on l_1, then either l_1 is perpendicular to l_2 or l_1 is perpendicular to l_3.

Next, due to the fact that the same configuration of P_1, P_2 and P_3 achieves both minima, we have

Corollary 4.4 Both the minimum point to line distance and the minimum edge length appear in the same right triangle where the shorter leg is a minimum edge and the altitude to the hypotenuse is the minimum point to line distance.

Corollary 4.4 can be verified by computing the distance from the intersection of the above two perpendicular lines to the third. The distance is the same as that in Theorem 4.1. Note that the coefficients the altitude of the hypotenuse are outside $[-L, L]$.

Furthermore, if we consider two distinct intersections of lines in S_L, then the minimum distance between such points is also known. Recall from the proof of Theorem 4.1 that in Table 4.3 the denominator of the minimum edge length follows the denominator of the minimum point to line distance. That is, the minimum edge length is in fact the second smallest point to line distance. This can be understood from Corollary 4.4 since the minimum edge length is the length of the shorter leg. Now, suppose there are two distinct intersection points: P is the intersection of l_1 and l_2 and Q is the intersection of l_3 and l_4. $|PQ|$ must be larger than the minimum point to line distance since the altitude of the hypotenuse in a minimum right triangle is not

in S_L. Because the two end points of a minimum edge are two distinct intersection points from S_L, therefore,

Corollary 4.5 The minimum distance between two distinct intersection points from S_L is equal to the length of a minimum edge.

4.4 3-D Minimum Point/Plane Distance and Minimum Edge

Let $p_i : a_i x + b_i y + c_i z + d_i = 0$, $i = 1, 2, 3, 4$, be four planes. Then the distance of the intersection of p_1, p_2 and p_3 from the plane p_4 is

$$\left| \frac{\begin{vmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{vmatrix}}{\sqrt{a_4^2 + b_4^2 + c_4^2} \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}} \right| \tag{4.5}$$

We call (4.5) the point to plane distance. Similarly, the distance between the intersections of p_1, p_2, p_3 and p_1, p_2, p_4 is

$$\left| \frac{\sqrt{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}^2 + \begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}^2 + \begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}^2} \begin{vmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_4 & b_4 & c_4 \end{vmatrix}} \right| \tag{4.6}$$

We consider(a_i, b_i, c_i) as a point P_i in the 3-D rectangular grid space $[-L, L] \times [-L, L] \times [-L, L]$. Then $\sqrt{a_i^2 + b_i^2 + c_i^2}$ is the length of segment OP_i and $\begin{vmatrix} a_i & b_i & c_i \\ a_j & b_j & c_j \\ a_k & b_k & c_k \end{vmatrix}$ is 6 times the signed volume of the tetrahedron $OP_iP_jP_k$. To simplify notation, the 4×4 determinant common to both numerators is denoted as $det(a, b, c, d)$.

4.4.1 Minimum Point to Plane Distance

Theorem 4.6 The minimum point to plane distance, that is, the minimum of (4.5), is

1) if L is multiple of 3,

$$\frac{1}{(4L^3 - L)\sqrt{3L^2 - 2L + 1}}$$

2) otherwise,

$$\frac{1}{(4L^3 - 5L + 2)\sqrt{3L^2 - 2L + 1}}$$

Proof: Analogous to the 2-D case, the numerator of (4.5) has to be 1 in order to minimize (4.5).

Following the approach in the 2-D case, we assume that $|a_i| \geq |b_i| \geq |c_i|$. The enumeration of $|OP_i|$ is shown in Table 4.4 and is obtained by assigning values to a_i, b_i and c_i in the order of Table 4.1.

The enumeration the 3×3 determinant in the descending order, on the other hand, is more difficult than the 2-D counterpart. One way is to evaluate the determinant in the order shown in the Table 4.5 (see Figure 4.8). The values in the last column in the table are sorted in total order. The first few largest values in the list are:

$$4L^3,\ 4L^3 - L,\ 4L^3 - 2L,\ 4L^3 - 3L + 1,\ 4L^3 - 3L,\ 4L^3 - 4L,\ 4L^3 - 5L + 2, \ldots$$

If more than two P_i chosen as $(\pm L, \pm L, \pm L)$, (4.5) will not be minimum, because,

Table 4.4 Descending attainable square root of $a_i^2 + b_i^2 + c_i^2$

a_i	b_i	c_i	$\sqrt{a_3^2 + b_3^2 + c_3^2}$
$\pm L$	$\pm L$	$\pm L$	$\sqrt{L^2 + L^2 + L^2}$
$\pm L$	$\pm L$	$\pm(L-1)$	$\sqrt{L^2 + L^2 + (L-1)^2}$
$\pm L$	$\pm L$	$\pm(L-2)$	$\sqrt{L^2 + L^2 + (L-2)^2}$
$\pm L$	$\pm(L-1)$	$\pm(L-1)$	$\sqrt{L^2 + (L-1)^2 + (L-1)^2}$
...	...	...	.....

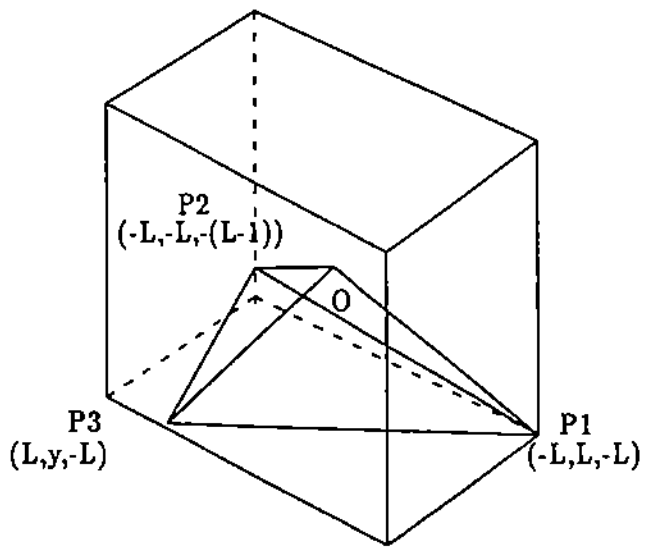

Figure 4.8 Arrangement of points P_1, P_2 and P_3

Table 4.5 Enumeration of the 3×3 determinant

P_1	P_2	P_3	$\lvert det(P_1, P_2, P_3)\rvert$
$(-L, L, -L)$	$(-L, -L, -L)$	$(L, y, -L)$	$4L^3$
$(-L, L, -L)$	$(-L, -L, -(L-1))$	$(L, y, -L)$	$4L^3 - L(L+y)$
$(-(L-1), L, -L)$	$(-L, -L, -(L-1))$	$(L, y, -L)$	$4L^3 - 2L(L+y) + y$
$(-(L-2), L, -L)$	$(-L, -L, -(L-1))$	$(L, y, -L)$	$4L^3 - 3L(L+y) + 2y$
$(-(L-2), L, -L)$	$(-L, -L, -(L-2))$	$(L, y, -L)$	$4L^3 - 4L(L+y) + 3y$
.....	.....	.....	.....

$$
\begin{aligned}
det(a,b,c,d) &= \begin{vmatrix} \pm L & \pm L & \pm L & d_i \\ \pm L & \pm L & \pm L & d_j \\ a_k & b_k & c_k & d_k \\ a_l & b_l & c_l & d_l \end{vmatrix} \\
&= -d_i \begin{vmatrix} \pm L & \pm L & \pm L \\ a_k & b_k & c_k \\ a_l & b_l & c_l \end{vmatrix} + d_j \begin{vmatrix} \pm L & \pm L & \pm L \\ a_k & b_k & c_k \\ a_l & b_l & c_l \end{vmatrix} \\
&\quad -d_k \begin{vmatrix} \pm L & \pm L & \pm L \\ a_3 & b_3 & c_3 \\ a_l & b_l & c_l \end{vmatrix} + d_l \begin{vmatrix} \pm L & \pm L & \pm L \\ \pm L & \pm L & \pm L \\ a_3 & b_3 & c_3 \end{vmatrix} \\
&= L\, f(a,b,c,d)
\end{aligned}
$$

By Table 4.5, the y value has to be negative and as close to $-L$ as possible to maximize both $|OP_4|$ and the 3×3 determinant. With one P_i as $(\pm L, \pm L, \pm L)$ and the other three as $(\times, \times, \times)$ where two ×s are $\pm L$ and the remaining one is $\pm(L-1)$, the candidate 3×3 determinant is either $4L^3 - L$ or $4L^3 - 3L + 1$ by having $y = -(L-1)$ in either the second row or the third row of the table. Then the largest possible

denominator of (4.5) is

$$(4L^3 - L)\sqrt{L^2 + L^2 + (L-1)^2} \tag{4.7}$$

if the second row is chosen and it is

$$(4L^3 - 3L + 1)\sqrt{L^2 + L^2 + L^2} \tag{4.8}$$

for the third row.

Though (4.8) is greater than (4.7), it is impossible, however, to pick d_i such that $|det(a,b,c,d)| = 1$ in that case. Since $P_4 = (\pm L, \pm L, \pm L)$ and d_i is at most $O(L^2)$, let $d_i = e_i L^2 + f_i L + g_i$ for $i = 1, \ldots, 4$. We show that no matter how signs for the coordinates of P_4 have been chosen,

$$\begin{vmatrix} -(L-1) & L & -L & e_1L^2 + f_1L + g1 \\ -L & -L & -(L-1) & e_2L^2 + f_2L + g_2 \\ L & -(L-1) & -L & e_3L^2 + f_3L + g_3 \\ \pm L & \pm L & \pm L & e_4L^2 + f_4L + g_4 \end{vmatrix} = \pm 1$$

is not possible for any e_i, f_i and g_i.

Let us first assume that $P_4 = (L, L, L)$. The expanded determinant becomes

$$\begin{aligned} &(4e_4 + 4e_2)L^5 + (4f_4 + 4f_2 + 2e_3 + 2e_1)L^4 \\ &+(4g_4 + 4g_2 + 2f_3 + 2f_1 - 3e_4 - e_3 - e_2 - e_1)L^3 \\ &+(2g_3 + 2g_1 - 3f_4 - f_3 - f_2 - f_1 + e_4)L^2 \\ &-(3g_4 + g_3 + g_2 + g_1 - f_4)L + g_4 = \pm 1 \end{aligned}$$

and a system of linear equations is thus derived:

$$\begin{aligned} &4e_4 + 4e_2 = 0 \\ &4f_4 + 4f_2 + 2e_3 + 2e_1 = 0 \\ &4g_4 + 4g_2 + 2f_3 + 2f_1 - 3e_4 - e_3 - e_2 - e_1 = 0 \\ &2g_3 + 2g_1 - 3f_4 - f_3 - f_2 - f_1 + e_4 = 0 \\ &3g_4 + g_3 + g_2 + g_1 - f_4 = 0 \\ &g_4 = \pm 1 \end{aligned}$$

As is easily verified that the rank of the coefficient matrix of the linear system is 5. Therefore, there is no solution. Replacing P_4 by $(L, L, -L)$, $(L, -L, L)$ and $(L, -L, -L)$, each results in a singular linear system that has no solution. Since these are the four distinct cases, no suitable d_i exists.

Now, suppose the second row is chosen. We show that there exist solutions for d_i such that $|det(a, b, c, d)| = 1$. Assume $P_1 = (-L, L, -L)$, $P_2 = (-L, -L, -(L-1))$, $P_3 = (L, -(L-1), -L)$, $P_4 = (L-1, L, L)$ (no common factor of L or $L-1$ in the first three columns) and $d_i = f_i L + g_i$. Then

$$\begin{vmatrix} -L & L & -L & f_1 L + g1 \\ -L & -L & -(L-1) & f_2 L + g_2 \\ L & -(L-1) & -L & f_3 L + g_3 \\ L-1 & L & L & f_4 L + g_4 \end{vmatrix} = \pm 1$$

Expanding the determinant, one gets

$$\begin{aligned} &(4f_4 + 4f_2)L^4 + (4g_4 + 4g_2 + 4f_3 - 2f_2 + 2f_1)L^3 \\ &+(4g_3 - 2g_2 + 2g_1 - f_4 - f_3 + f_2 - 3f_1)L^2 \\ &-(g_4 + g_3 - g_2 + 3g_1 - f_1)L - g_1 = \pm 1 \end{aligned}$$

The corresponding system of linear equations is

$$\begin{aligned} &4f_4 + 4f_2 = 0 \\ &4g_4 + 4g_2 + 4f_3 - 2f_2 + 2f_1 = 0 \\ &4g_3 - 2g_2 + 2g_1 - f_4 - f_3 + f_2 - 3f_1 = 0 \\ &g_4 + g_3 - g_2 + 3g_1 - f_1 = 0 \\ &g_1 = \pm 1 \end{aligned}$$

with the solution

$$\begin{aligned} f_1 &= g_4 + g_3 - g_2 + 3g_1 \\ f_2 &= \frac{3g_4 - 3g_3 - 3g_2 + 11g_1}{3} \\ f_3 &= -\frac{3g_4 + 3g_3 + 3g_2 - g_1}{3} \end{aligned}$$

$$f_4 = -\frac{3g_4 - 3g_3 - 3g_2 + 11g_1}{3}$$

Let $g_1 = -1 = g_2 = g_3 = -1$ and $g_4 = 1$, then $f_1 = -2$, $f_2 = -\frac{2}{3}$ and $f_3 = f_4 = \frac{2}{3}$. So, $d_1 = -2L - 1$, $d_2 = -\frac{2}{3}L - 1$, $d_3 = \frac{2}{3}L - 1$ and $d_4 = \frac{2}{3}L + 1$. Clearly $|d_i| < 3L^2$ and if L is multiple of 3, all the d_is are integers. Hence, this completes the case of L is multiple of 3.

If L is not a multiple of 3, then a detailed case analysis is needed to find minimum of (4.5). Based on the necessary condition that $|det(a, b, c, d)| = 1$, we can narrow our choices down to only a few candidate selections of the P_is.

Let $d_i = e_iL^2 + f_iL + g_i$ and $s_i = \pm 1$ as sign variable. We have to analyze row by row of P_i starting from the second row of the table. Using the second row as an example, if $P_4 = (\pm(L-1), \pm L, \pm L)$, then from

$$\begin{vmatrix} -L & L & -L & e_1L^2 + f_1L + g1 \\ -L & -L & -(L-1) & e_2L^2 + f_2L + g_2 \\ L & y & -L & e_3L^2 + f_3L + g_3 \\ s_1(L-1) & s_2L & s_3L & e_4L^2 + f_4L + g_4 \end{vmatrix} = \pm 1$$

one gets

$$\begin{aligned} &(s_3(-2e_3 - e_2 - e_1) + s_2(2e_1 - 2e_2) + s_1(2e_3 - e_2 - e_1) - 4e_4)L^5 \\ &+(s_3((e_1 - e_2)y - 2f_3 - f_2 - f_1) + s_2(-2f_2 + 2f_1 - e_3 - e_1) \\ &+s_1((e_2 - e_1)y + 2f_3 - f_2 - f_1 - 3e_3 + e_2 + e_1) - 4f_4 + e_4)L^4 \\ &+(s_3((f_1 - f_2)y - 2g_3 - g_2 - g_1) + s_2(-2g_2 + 2g_1 - f_3 - f_1) \\ &+s_1((f_2 - f_1 - e_2 + 2e_1)y + 2g_3 - g_2 - g_1 - 3f_3 + f_2 + f_1 + e_3) \\ &+e_4y - 4g_4 + f_4)L^3 \\ &+(s_3(g_1 - g_2)y + s_2(-g_3 - g_1) + s_1((g_2 - g_1 - f_2 + 2f_1 - e_1)y \\ &-3g_3 + g_2 + g_1 + f_3) + f_4y + g_4)L^2 \\ &-(s_1((-g_2 + 2g_1 - f_1)y + g_3) + g_4y)L \\ &-s_1g_1y = \pm 1 \end{aligned}$$

Since $L > 1$, $|g_1| < L$ and $|y| \leq L$, it is necessary that $g_1 = \pm 1$ and $y = \pm 1$ or $y = \pm(L-1)$. The same conditions are derived if P_4 is replaced by $(\pm(L-1), \pm L, \pm(L-1))$,

or $(\pm(L-1), \pm(L-1), \pm L)$, or $(\pm(L-1), \pm(L-1), \pm(L-1))$. We do not need to pursue further coordinate combinations for P_4 since those values will not result in large enough denominator so as to minimize (4.5).

Similar necessary conditions are obtained for each successive row of Table 4.5. After substituting $y = -(L-2)$ into the second row, and picking $P_4 = (L, L-1, L)$, a solvable system of linear equations is derived with integer solutions. Therefore, the minimum of (4.5) when L is not multiple of 3 is

$$\frac{1}{(4L^3 - 5L + 2)\sqrt{3L^2 - 2L + 1}}$$

with $P_1 = (-(L-1), L, -L)$, $P_2 = (-L, -L, -(L-1))$, $P_3 = (L, -(L-2), -L)$, $P_4 = (L, L-1, L)$ and $d_1 = -1$, $d_2 = 3$, $d_3 = 5$ and $d_4 = -3$. The 3×3 determinant in this case is $4L^3 - 5L + 2$ □

4.4.2 Minimum Edge Length

In analogy to the 2-D case, (4.6) can be rewritten as

$$\left| \frac{\begin{vmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{vmatrix}}{\sqrt{a_4^2 + b_4^2 + c_4^2}\, \sin \angle(OP_4, P_1OP_2) \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}} \right| \tag{4.9}$$

where $\angle(OP_4, P_1OP_2)$ is the angle between OP_4 and the plane containing P_1, O, P_2. Hence, $\sqrt{a_4^2 + b_4^2 + c_4^2}\, \sin \angle(OP_4, P_1OP_2)$ is in fact the altitude of P_4 from the plane containing O, P_1, P_2.

Theorem 4.7 The minimum edge length is

1) if L is multiple of 3,

$$\frac{\sqrt{3L^2-2L+1}}{(2L-1)\sqrt{2L^2+1}(4L^3-L^2+L)}$$

2) otherwise,

$$\frac{\sqrt{2}\sqrt{3L^2+1}}{(4L^2-L-1)(4L^3-L^2+L)}$$

Proof: Geometrically, for three points P_1, P_2 and P_4, the maximum distance from P_4 to the plane containing O, P_1, P_2 can be $\sqrt{3}L$. This happens at the arrangement shown in Figure 4.9. Hence, to have a long distance between P_4 and OP_1P_2, P_1 and P_2 should be chosen close to both the $x=0$ and $y=0$ planes as depicted in Figure 4.9. However, the value of the 3×3 determinant of P_1, P_2 and P_3 would be small if P_1 and P_2 both are close to the $x=0$ and $y=0$ planes no matter how P_3 is chosen. The value of the 3×3 determinant is larger when all the three points are toward the corner of the grid cube as shown in the proof of the last theorem. Therefore, a balanced selection of P_i is needed to make (4.6) minimum.

Because of the constraint that the numerator has to be 1 in order to minimize (4.6), we only look at those selections that meet this necessary condition. From the proof of the theorem 4.6, we know that y in Figure 4.8 has to be either $\pm(L-1)$ or ± 1. Thus, only values obtained by letting $y=\pm(L-1)$ and $y=\pm 1$ are compared with each other. With a careful enumeration of possible such values, we obtain the following

- L is multiple of 3:

$$\frac{\sqrt{3L^2-2L+1}}{(2L-1)\sqrt{2L^2+1}(4L^3-L^2+L)} \tag{4.10}$$

  with $P_1=(-(L-1),L,-L)$, $P_2=(L,-1,-L)$, $P_3=(-L,-L,-L)$, $P_4=(L,L,L-1)$ and $d_1=-\frac{2}{3}L$, $d_2=-\frac{2}{3}L$, $d_3=-2L-1$ and $d_4=2L$.

- L is not multiple of 3:

$$\frac{\sqrt{2}\sqrt{3L^2+1}}{(4L^2-L-1)(4L^3-L^2+L)} \tag{4.11}$$

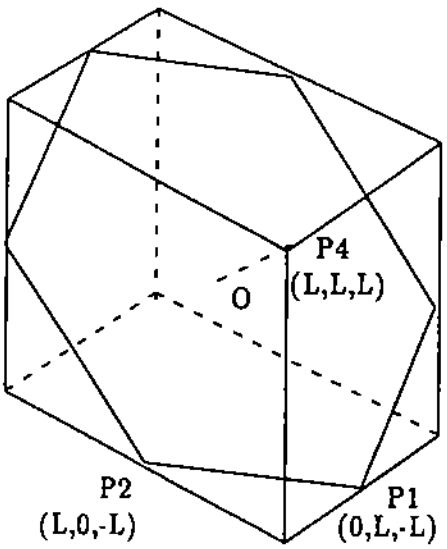

Figure 4.9 Maximum distance from P_4 to the plane containing O, P_1 and P_2

with $P_1 = (-L, L, -L)$, $P_2 = (L, -1, -L)$, $P_3 = (-L, -L, -(L-1))$, $P_4 = (L-1, L, L)$ and $d_1 = -1$, $d_2 = 0$, $d_3 = -1$ and $d_4 = 1$.

These are the results of the theorem □

4.5 n-D Minimum Point/Plane Distance and Minimum Edge

Let $p_i : a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n + a_{i(n+1)} = 0$, $i = 1, 2, \ldots, n+1$ be $n+1$ hyperplanes in the $\mathbf{E}^n$ where $n > 3$. Then, the distance from the intersection of p_1, $p_2, \ldots, p_n$ to p_{n+1} is

$$\left| \frac{\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1(n+1)} \\ a_{21} & a_{22} & \cdots & a_{2(n+1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n(n+1)} \\ a_{(n+1)1} & a_{(n+1)2} & \cdots & a_{(n+1)(n+1)} \end{vmatrix}}{\sqrt{\sum_{i=1}^{n} a_{(n+1)i}^2} \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}} \right| \tag{4.12}$$

The minimum edge length formula is similar to those in the 2-D and 3-D cases. The following definition is used to simplify the notation for expressing the n-D edge length formula.

Definition. Let M be a $m \times n$ matrix where $m < n$. Then the square sum of minors of M, $S^2_{(m,n)}(M)$, is

$$\sum_{i_1, i_2, \ldots, i_m} |M_{i_1, i_2, \ldots, i_m}|^2$$

where $M_{i_1, i_2, \ldots, i_m}$ is the square submatrix of M consisting of the i_1th, i_2th, $\ldots$, i_mth columns of M.

Lemma 4.8 The distance between the intersection of $p_1, p_2, \ldots, p_{n-1}, p_n$ and the intersection of $p_1, p_2, \ldots, p_{n-1}, p_{n+1}$ is

$$\frac{\left|\begin{array}{c} \\ \sqrt{S^2_{(n-1,n)}(M_c)} \\ \\ \end{array}\begin{array}{|cccc|} a_{11} & a_{12} & \cdots & a_{1(n+1)} \\ a_{21} & a_{22} & \cdots & a_{2(n+1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n(n+1)} \\ a_{(n+1)1} & a_{(n+1)2} & \cdots & a_{(n+1)(n+1)} \end{array}\right|}{\left\|\begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1)1} & a_{(n-1)2} & \cdots & a_{(n-1)n} \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right\| \left\|\begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1)1} & a_{(n-1)2} & \cdots & a_{(n-1)n} \\ a_{(n+1)1} & a_{(n+1)2} & \cdots & a_{(n+1)n} \end{array}\right\|} \tag{4.13}$$

where M_c is the coefficient matrix of the first $n-1$ hyperplane equations.

Proof: Let $x_i^{\{1,2,\ldots,n-1,n\}}$ and $x_i^{\{1,2,\ldots,n-1,n+1\}}$ be the ith coordinate of the intersections of $p_1, p_2, \ldots, p_{n-1}, p_n$ and $p_1, p_2, \ldots, p_{n-1}, p_{n+1}$. The distance between the two intersections is

$$\sqrt{\sum_{i=1}^{n}(x_i^{\{1,2,\ldots,n-1,n\}} - x_i^{\{1,2,\ldots,n-1,n+1\}})^2}$$

Let Δ^a be the determinant of the coefficient matrix of $p_1, p_2, \ldots, p_{n-1}, p_n$ and Δ_i^a be Δ^a with the ith column replaced by the column of the negative constant terms of $p_1, p_2, \ldots, p_{n-1}, p_n$. Δ^b and Δ_i^b are similarly defined for $p_1, p_2, \ldots, p_{n-1}, p_{n+1}$. Then

$$x_i^{\{1,2,\ldots,n-1,n\}} = \frac{\Delta_i^a}{\Delta^a}$$

and

$$x_i^{\{1,2,\ldots,n-1,n+1\}} = \frac{\Delta_i^b}{\Delta^b}$$

Hence, the distance can be written as

$$\frac{\sqrt{\sum_{i=1}^{n}(\Delta_i^a\Delta^b - \Delta^a\Delta_i^b)^2}}{|\Delta^a\Delta^b|}$$

Denote M_c^i the square submatrix of M_c consisting of all the columns of M_c except the ith column and D the $(n+1)\times(n+1)$ matrix in the numerator. It is sufficient to show that

$$|\Delta_i^a\Delta^b - \Delta^a\Delta_i^b| = \text{abs}(|M_c^i|D)$$

Without loss of generality, let $i = 1$. Then we need to show

$$\text{abs}\left(\begin{vmatrix} -a_{1(n+1)} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ -a_{(n-1)(n+1)} & a_{(n-1)2} & \cdots & a_{(n-1)n} \\ -a_{n(n+1)} & a_{n2} & \cdots & a_{nn} \end{vmatrix} \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1)1} & a_{(n-1)2} & \cdots & a_{(n-1)n} \\ a_{(n+1)1} & a_{(n+1)2} & \cdots & a_{(n+1)n} \end{vmatrix}\right.$$

$$\left. - \begin{vmatrix} -a_{1(n+1)} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ -a_{(n-1)(n+1)} & a_{(n-1)2} & \cdots & a_{(n-1)n} \\ -a_{(n+1)(n+1)} & a_{(n+1)2} & \cdots & a_{(n+1)n} \end{vmatrix} \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1)1} & a_{(n-1)2} & \cdots & a_{(n-1)n} \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}\right)$$

$$= \text{abs}\left(\begin{vmatrix} a_{12} & \cdots & a_{1n} \\ a_{22} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{(n-1)2} & \cdots & a_{(n-1)n} \end{vmatrix} \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1(n+1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n(n+1)} \\ a_{(n+1)1} & a_{(n+1)2} & \cdots & a_{(n+1)(n+1)} \end{vmatrix}\right) \qquad (4.14)$$

Using the Lapalace rule, the left side of (4.14) is the same as

$$\text{abs}\left(\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} & & & & \\ \vdots & \vdots & \ddots & \vdots & & & & \\ a_{(n-1)1} & a_{(n-1)2} & \cdots & a_{(n-1)n} & & & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} & -a_{n(n+1)} & a_{n2} & \cdots & a_{nn} \\ a_{(n+1)1} & a_{(n+1)2} & \cdots & a_{(n+1)n} & -a_{(n+1)(n+1)} & a_{(n+1)2} & \cdots & a_{(n+1)n} \\ & & & & -a_{1(n+1)} & a_{12} & \cdots & a_{1n} \\ & & & & \vdots & \vdots & \ddots & \vdots \\ & & & & -a_{(n-1)(n+1)} & a_{(n-1)2} & \cdots & a_{(n-1)n} \end{vmatrix}\right) \qquad (4.15)$$

Subtracting the ith column from the $(n+i)$th column, for $i = 2, \ldots, n$, (4.15) is equal to

$$\text{abs}\left(\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} & & -a_{12} & \cdots & -a_{1n} \\ \vdots & \vdots & \ddots & \vdots & & -a_{22} & \cdots & -a_{2n} \\ a_{(n-1)1} & a_{(n-1)2} & \cdots & a_{(n-1)n} & & -a_{(n-1)2} & \cdots & -a_{(n-1)n} \\ a_{n1} & a_{n2} & \cdots & a_{nn} & -a_{n(n+1)} & 0 & \cdots & 0 \\ a_{(n+1)1} & a_{(n+1)2} & \cdots & a_{(n+1)n} & -a_{(n+1)(n+1)} & 0 & \cdots & 0 \\ & & & & -a_{1(n+1)} & a_{12} & \cdots & a_{1n} \\ & & & & \vdots & \vdots & \ddots & \vdots \\ & & & & -a_{(n-1)(n+1)} & a_{(n-1)2} & \cdots & a_{(n-1)n} \end{vmatrix}\right) \quad (4.16)$$

Then adding the $(n+1+i)$th row to the ith row, for $i = 1, \ldots, n-1$, (4.16) is the same as

$$\text{abs}\left(\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} & -a_{1(n+1)} & & & \\ \vdots & \vdots & \ddots & \vdots & \vdots & & & \\ a_{(n-1)1} & a_{(n-1)2} & \cdots & a_{(n-1)n} & -a_{(n-1)(n+1)} & & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} & -a_{n(n+1)} & & & \\ a_{(n+1)1} & a_{(n+1)2} & \cdots & a_{(n+1)n} & -a_{(n+1)(n+1)} & & & \\ & & & & -a_{1(n+1)} & a_{12} & \cdots & a_{1n} \\ & & & & \vdots & \vdots & \ddots & \vdots \\ & & & & -a_{(n-1)(n+1)} & a_{(n-1)2} & \cdots & a_{(n-1)n} \end{vmatrix}\right) \quad (4.17)$$

Expanding (4.17), the result is precisely the right side of (4.14) □

We do not further pursue the minimum point/plane distance and minimum edge length because of the complexity associated with the $n \times n$ determinants. In fact, the problem to determine the order of $n \times n$ determinants whose elements are bounded integers is itself an open problem.

4.6 Lower Bound of Precision for Classification

As Sugihara and Iri pointed out in [33], the sign of (4.1) or (4.5) is sufficient to determine where a point lies with respect to a line or a plane. The sufficient precision

for this process is the largest possible value for evaluating both the denominator and numerator. Since this value does not exceed $8L^4$ for 2-**D** or $48L^5$ for 3-D, it means we need roughly 4 times the input precision for point/line classification and 5 times the input precision for point/plane classification.

Now, because the exact point/line and point/plane distance are of $O(L^{-3})$ and $O(L^{-4})$ respectively, it is necessary that coordinates of the intersection points be computed with precision 3 times greater than the input precision for 2-**D** and 4 times greater than the input precision for 3-**D**. Then, the classification is done by substituting thus obtained intersection coordinates into the line or plane equation and evaluating the sign of the expression. The evaluation requires an extra $O(\log\ L)$ input precision.

Therefore, this necessary amount of precision we have derived matches the sufficient amount of precision of Sugihara and Iri [33]. Hence, we conclude the necessary and sufficient amount of precision for point/line and point/plane classifications are 4 and 5 times the input precision respectively. This bound is optimal.

5. POINT/CONIC AND POINT/QUADRIC CLASSIFICATION

5.1 Introduction

Having treated lines and planes, we treat the next simplest curves and surfaces: conics and quadrics. Conics and quadrics have been studied extensively for centuries. They are widely used today in computer-aided geometric design as degree two Rational Bézier curves/surfaces or degree two Rational B-Spline curves/surfaces. The point/conic or point/quadric classification problem is to determine to which side of a given conic or quadric a point lies. The problem is a fundamental geometric computation.

We study the precision required for exact point/conic or point/quadric classification. As in the linear counterpart, the point considered here is defined as the intersection of given conics or quadrics. Our results show that exact classification is always possible but that the precision required is very high. Because of this expensive precision requirement, it seems that exact classification is of little use in practice. From a more practical point of view, however, it seems that methods based on linear approximation are feasible. Although the point to be classified is no longer the original point but an approximation, the classification is still meaningful as long as the integrity of the approximated object is maintained.

5.2 Background and Notations

Let L be a positive integer. The class of general conics, C_L, under consideration is

$$C : ax^2 + bxy + cy^2 + dx + ey + f = 0 \tag{5.1}$$

where the coefficients of C are integers and satisfy the constraints: $-L \leq a, b, c \leq L$, $-L^2 \leq d, e \leq L^2$ and $-5L^3 \leq f \leq 5L^3$. Similarly, the class of general quadrics, Q_L, under consideration is

$$Q : ax^2 + by^2 + cz^2 + dxy + eyz + fzx + gx + hy + iz + j = 0 \qquad (5.2)$$

where the coefficients of Q are integers and meet the constraints: $-L \leq a, b, c, d, e, f \leq L$, $-L^2 \leq g, h, i \leq L^2$ and $-9L^3 \leq j \leq 9L^3$.

An intersection point for our purpose is defined as the intersection of at least two distinct conics from C_L or three distinct quadrics from Q_L. Because of the coefficient constraints, either C_L or Q_L contains only finitely many conics or quadrics respectively. Hence, the number of intersection points defined by C_L or Q_L is also finite.

Note that two conics intersect generally in four points and three quadrics intersect generally in eight points. We do not present any method here to isolate one of these intersection points. Rather, since the amount of precision required for the classification remains the same as long as the conics or quadrics are all from either C_L or Q_L, we study the problem without isolating any point. Of course, there are special cases where intersection point(s) can be classified with respect to the input conic or quadrics by using less precision.

5.3 Exact Method

The mathematical tool used for the exact classification is the polynomial resultant from elimination theory. The resultant of two univariate polynomial equations

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 = 0$$

and

$$g(x) = b_m x^m + b_{m-1} x^{m-1} + \cdots + b_0 = 0$$

is the following $(n+m)\times(n+m)$ determinant

$$Res_x(f,g) = \begin{vmatrix} a_n & a_{n-1} & \cdots & a_0 & 0 & \cdots & 0 \\ 0 & a_n & \cdots & a_1 & a_0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & 0 & a_n & a_{n-1} & \cdots & a_0 \\ b_m & b_{m-1} & \cdots & b_0 & 0 & \cdots & 0 \\ 0 & b_m & \cdots & b_1 & b_0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & 0 & b_m & b_{m-1} & \cdots & b_0 \end{vmatrix}$$

If $Res_x(f,g) = 0$, then $f(x) = 0$ and $g(x) = 0$ must have a common root. The reader is referred to [36] for detailed account on the subject.

5.3.1 Point/Conic Classification

Specifically, let $C_i : a_i x^2 + b_i xy + c_i y^2 + d_i x + e_i y + f_i = 0$, $i = 1,2,3$, be three conics from C_L. $P(x_0, y_0)$ is one of the intersection points of C_1 and C_2. Our task is to determine the minimum precision required to classify P with respect to C_3.

The sign of the expression after substituting P into C_3 classifies P exactly with respect to C_3. Let

$$D = a_3 x_0^2 + b_3 x_0 y_0 + c_3 y_0^2 + d_3 x_0 + e_3 y_0 + f_3 \tag{5.3}$$

Then the problem is to determine the sign of D. Since P is one of the four intersections of C_1 and C_2, P must be on both curves and that means,

$$a_1 x_0^2 + b_1 x_0 y_0 + c_1 y_0^2 + d_1 x_0 + e_1 y_0 + f_1 = 0 \tag{5.4}$$

and

$$a_2 x_0^2 + b_2 x_0 y_0 + c_2 y_0^2 + d_2 x_0 + e_2 y_0 + f_2 = 0 \tag{5.5}$$

We eliminate x_0 and y_0 from equations (5.4), (5.5) and (5.3). Thereby, a univariate polynomial $F(D)$ is derived. Next, we construct a negative polynomial remainder

sequence, which is a Sturm sequence, for $F(D)$ and $F'(D)$. The sign of D can be determined by isolating the real roots of $F(D)$ into separate intervals using the sequence [39].

To compute $F(D)$, we apply resultants to C_1, C_2 and C_3 to first eliminate x_0 and then y_0. Let

$$\begin{aligned} f_{12}(y_0) &= Res_{x_0}((5.4),(5.5)) \\ &= \begin{vmatrix} a_1 & b_1y_0+d_1 & c_1y_0^2+e_1y_0+f_1 & 0 \\ 0 & a_1 & b_1y_0+d_1 & c_1y_0^2+e_1y_0+f_1 \\ a_2 & b_2y_0+d_2 & c_2y_0^2+e_2y_0+f_2 & 0 \\ 0 & a_2 & b_2y_0+d_2 & c_2y_0^2+e_2y_0+f_2 \end{vmatrix} \\ &= A_4y_0^4+A_3y_0^3+A_2y_0^2+A_1y+A_0 \end{aligned} \tag{5.6}$$

where $A_i = O(L^{8-i})$, for $i = 4,\ldots,0$ and

$$\begin{aligned} f_{13}(y_0) &= Res_{x_0}((5.4),(5.3)) \\ &= \begin{vmatrix} a_1 & b_1y_0+d_1 & c_1y_0^2+e_1y_0+f_1 & 0 \\ 0 & a_1 & b_1y_0+d_1 & c_1y_0^2+e_1y_0+f_1 \\ a_3 & b_3y_0+d_3 & c_3y_0^2+e_3y_0+f_3-D & 0 \\ 0 & a_3 & b_3y_0+d_3 & c_3y_0^2+e_3y_0+f_3-D \end{vmatrix} \\ &= B_4y_0^4+B_3y_0^3+(B_2+M_1D)y_0^2+(B_1+M_1'D)y+B_0+M_1''D+M_2''D^2 \end{aligned} \tag{5.7}$$

where $B_i = O(L^{8-i})$, for $i = 4,\ldots,0$ and $M_1 = O(L^3)$, $M_1' = O(L^4)$, $M_1'' = O(L^5)$ and $M_2'' = O(L^2)$. Now, let $g_2(D) = B_2 + M_1D$, $g_1(D) = B_1 + M_1'D$ and $g_0(D) = B_0 + M_1''D + M_2''D^2$, then

$$F(D) = Res_{y_0}(f_{12}(y_0), f_{13}(y_0))$$

$$= \begin{vmatrix} A_4 & A_3 & A_2 & A_1 & A_0 & 0 & 0 & 0 \\ 0 & A_4 & A_3 & A_2 & A_1 & A_0 & 0 & 0 \\ 0 & 0 & A_4 & A_3 & A_2 & A_1 & A_0 & 0 \\ 0 & 0 & 0 & A_4 & A_3 & A_2 & A_1 & A_0 \\ B_4 & B_3 & g_2(D) & g_1(D) & g_0(D) & 0 & 0 & 0 \\ 0 & B_4 & B_3 & g_2(D) & g_1(D) & g_0(D) & 0 & 0 \\ 0 & 0 & B_4 & B_3 & g_2(D) & g_1(D) & g_0(D) & 0 \\ 0 & 0 & 0 & B_4 & B_3 & g_2(D) & g_1(D) & g_0(D) \end{vmatrix}$$

$$= g_8 D^8 + g_7 D^7 + g_6 D^6 + g_5 D^5 + g_4 D^4 + g_3 D^3 + g_2 D^2 + g_1 D + g_0 \qquad (5.8)$$

where $g_i = O(L^{48-3i})$, for $i = 8, \ldots, 0$.

Now, the precision needed to construct a negative polynomial remainder sequence, which is a Sturm sequence, is $O(L^{48\text{-}15})$ [11]. Furthermore, we need more precision when we actually want to isolate the real roots of $F(D)$ by a set of nonintersecting real intervals using the root separation estimate given by Mignotte [21].

Clearly, this approach solves the classification problem precisely without any ambiguity. In the course of implementing geometric algorithms where topological decision is based on numeric computations, such exact classification is necessary to preserve the correct topological structure of the geometric objects under computation. When topological data is given a higher priority than numerical data, exact methods eliminates the potential topological violations that might be caused by numeric computations.

5.3.2 Point/Quadric Classification

The resultant method easily generalizes to classifying a point with respect to a given quadric. In fact, the method generalizes naturally to more general classification problems such as point/cubics, etc, in a similar setting. The procedure is exactly the same while the precision required will be more.

Let $Q_i(x, y, z)$, $i = 1, 2, 3, 4$, be four quadrics from Q_L and $P(x_0, y_0, z_0)$ be one of the intersections of Q_1, Q_2 and Q_3. To classify P with respect to Q_4, we simply

determine the sign of

$$D = Q_4(x_0, y_0, z_0)$$

Because $Q_i(x_0, y_0, z_0) = 0$ for $i = 1, 2, 3$, we form the following resultants:

$$\begin{aligned}
f_{12}(y_0, z_0) &= Res_{x_0}(Q_1(x_0, y_0, z_0), Q_2(x_0, y_0, z_0)) \\
f_{13}(y_0, z_0) &= Res_{x_0}(Q_1(x_0, y_0, z_0), Q_3(x_0, y_0, z_0)) \\
f_{14}(y_0, z_0) &= Res_{x_0}(Q_1(x_0, y_0, z_0), Q_4(x_0, y_0, z_0) - D) \\
f_{123}(z_0) &= Res_{y_0}(f_{12}(y_0, z_0), f_{13}(y_0, z_0) \\
f_{124}(z_0) &= Res_{y_0}(f_{12}(y_0, z_0), f_{14}(y_0, z_0) \\
F(D) &= Res_{z_0}(f_{123}(z_0), f_{124}(z_0))
\end{aligned}$$

Then, $F(D)$ is a polynomial of degree 128 with coefficients of order 1280 times the input precision. One needs roughly $1280 \cdot 255$ times the input precision just to construct a Sturm sequence for $F(D)$.

5.3.3 Remark on the Exact Method

The obvious drawback of the exact method is that it needs too much precision. Take the resultant of two degree d polynomials with their leading coefficients representable in $O(L^l)$ and constant terms in $O(L^c)$, then the result is a degree d^2 polynomial whose constant term may grow as large as $O(L^{cd+ld})$. Therefore, $F(D)$ grows exponentially in terms of both the degree and coefficients. See Table 5.1 for a few examples.

Along with such a high precision requirement, the computational cost of carrying out the resultants calculation is also excessive. However, the procedure outlined above only gives upper bounds on the precision required. To illustrate that the bounds are not sharp, we look at the simpler problem of 2-D point/line classification by using the method. Suppose $l_i : a_i x + b_i y + c_i = 0$, $i = 1, 2, 3$, are three given lines with a_i, b_i and c_i are integers and $-L \leq a_i, b_i \leq L$ and $-2L^2 \leq c_i \leq 2L^2$. We want to classify the intersection of l_1 and l_2 with respect to l_3. By eliminating x and y following the

Table 5.1 Degree and coefficient growth

# of Vars Eliminated	Lead Coeff in $O(L)$	Deg of Polys	Deg of $F(D)$	Constant Term in $O(L)$
0	1	d	1	$d+1$
1	$2d$	d^2	d	$d(d+1)+d = d(d+2)$
2	$4d^3$	d^4	d^3	$(d(d+2))d^2 + 2d \cdot d^2 = d^3(d+4)$
3	$8d^7$	d^8	d^7	$(d^3(d+4))d^4 + 4d^3 \cdot d^4 = d^7(d+8)$
4	$16d^{15}$	d^{16}	d^{15}	$(d^7(d+8))d^8 + 8d^7 \cdot d^8 = d^{15}(d+16)$

procedure, we have

$$F(D) = -a_1 \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} D + a_1 \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} = g_1 D + g_0$$

Apparently, $a_1 = O(L)$ is the *gcd* of g_1 and g_0. Since the sign of a_1 is known, it is sufficient to judge the sign of the roots of the reduced polynomial whose coefficients are relatively prime. Note that the resultant method does not provide us with any information regarding the *gcd* of the coefficients of $F(D)$. As a matter of fact, the *gcd* of the coefficients of $F(D)$, measured in terms of the order of magnitude of L, may also grow exponentially as a function of the dimension. Again, take the example of point/hyperplane (line in 2-D) classification where the point is the intersection of n hyperplanes (two lines if $n = 2$) in n-D and the coefficients of the hyperplanes are subject to the constraints as described in the previous chapter. Let $f(n)$ denote the *gcd* of the coefficients of $F(D)$ in terms of the order of magnitude of L. Then $f(n)$ satisfies the following recurrence:

$$f(n) = \begin{cases} L & \text{if } n = 2 \\ f^2(n-1)L^{n-1} & \text{if } n > 2 \end{cases}$$

To solve the recurrence, we have

$$\begin{aligned} f(n) &= f^2(n-1)L^{n-1} = f^{2^2}(n-2)L^{(n-1)+2(n-2)} \\ &= \cdots\cdots\cdots\cdots \\ &= L^{(n-1)+2(n-2)+2^2(n-3)+\cdots+2^{n-3}2+2^{n-2}} \\ &= L^{2^n-n-1} \end{aligned}$$

This expression for $f(n)$ verifies our earlier results on point/line, point/hyperplane classifications by using exact point to line and point to hyperplane distance computations. Using that method, we showed that with dimension increased by one, the precision required increased only by a factor of L. The resultant approach, on the other hand, doubles the coefficient each time a resultant is taken. This is the reason why $f(n)$ is order of L^{2^n}.

We expect a similar recurrence relation of the *gcd* of the coefficients of $F(D)$ to exist in the case of classifying point/conics, point/quadrics and etc. by using the method. Furthermore, we would conjecture that if there is such a function, it may grow even faster than its linear counterpart. However, we are not able to confirm such information as the explicit expression of $F(D)$ in these cases gets too complex to be handled by our current available computing resources.

5.4 Approximation Method

In this section, we estimate the precision needed for using approximation methods to solve the classification problem. Of course, with conics and quadrics being lowest degree curves and curved surfaces, their approximants have to be lines and planes. Hence, after approximation, the problem is reduced to the point/line or point/plane classification problem. Applying results from the previous chapter, the required precisions are thus obtained.

There are many different methods to piecewise linearly approximate conic curves and quadrics surfaces. Therefore, to study the needed precision, we should not make our analysis dependent on any specific approximation method. Since an approximation method will generate a set of points that is within a given tolerance of the conic or quadric depending on the precision imposed, we can derive suitable bounds for the classification problem. From now on, we denote ϵ as the given tolerance.

5.4.1 Point/Conic Classification

Assume that after applying a suitable approximation method, the conic in question is given by a set of points defining the set of approximating line segments such that for any point on the curve C, there is a corresponding line segment and the distance from the point to the line segment in y direction is within ϵ. Furthermore, we assume that these points lie in the region of $[-M, M] \times [-M, M]$ and the second order derivative of y with respect to x is bounded by K in $[-M, M] \times [-M, M]$.

The coordinates of these points may consist of both the integral part and the fractional part. Hence, $\log_2 M - \log_2 \epsilon$ amount of precision ($\log_2 \epsilon$ is negative) is necessary to represent any coordinates of the points.

The equation of the line passing through two such consecutive points: $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ is

$$\begin{vmatrix} x & y & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix} = 0$$

or equivalently,

$$\begin{vmatrix} y_1 & 1 \\ y_2 & 1 \end{vmatrix} x - \begin{vmatrix} x_1 & 1 \\ x_2 & 1 \end{vmatrix} y + \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} = 0$$

Thus, $\log_2 M - \log_2 \epsilon + 1$ and $2(\log_2 M - \log_2 \epsilon) + 1$ amount of precisions are necessary to represent the linear term coefficient and constant term of the above line equation respectively. We may think of the coefficients of the line equation as integers with a sufficiently large scale factor. Since the approximation replaces all the three conics by sets of line segments with linear term coefficient representable in at least $\log_2 M - \log_2 \epsilon + 1$ amount of precision, using our result on the point/line classification, we need at least

$$4(\log_2 M - \log_2 \epsilon + 1)$$

amount of precision to classify the point and the approximated conic relationship.

Now, we have to account for the precision needed to actually compute a piecewise linear approximation that satisfies our initial assumption. The minimum precision estimate depends on the method of approximation. Later, we will estimate the precision needed to classify points with respect to the approximated curve.

To show the minimum precision needed to compute a piecewise linear approximation, we solve y from C in terms of x.

$$y = \frac{-(bx + e) \pm \sqrt{(b^2 - 4ac)x^2 + (2be - 4cd)x + e^2 - 4cf}}{2c} \tag{5.9}$$

From linear interpolation theory, we know that if $f(t)$ is a real function in C^2 on $[a,b]$ and $l(t)$ is the line segment joining $(a, f(a))$ and $(b, f(b))$, then for all $t \in [a,b]$,

$$|f(t) - l(t)| \leq \frac{(b-a)^2}{8} \max_{t \in [a,b]} |f''(t)|$$

Let δ be the x step size when using (5.9) to compute the approximation. Then

$$\frac{\delta^2}{8} K \leq \epsilon$$

so

$$\delta \leq \sqrt{\frac{8\epsilon}{K}}$$

We choose a positive δ. Now the following simple loop computes a piecewise linear approximation.

```
assign x_0 an initial value;
evaluate y_0 by (5.9)
for i = 1 to N do
    x_i = x_{i-1} + δ;
    evaluate y_i by (5.9);
endfor
```

where x_0 is the starting point and N is the total number of points. Since K generally is not a very big number, we assume that

$$\epsilon \leq \frac{8}{K}$$

or

$$\epsilon^2 \leq \frac{8\epsilon}{K}$$

hence,

$$\epsilon \leq \delta$$

Therefore, if we start with x_0 in $\log_2 M - \log_2 \epsilon$ amount of precision, then $x = x + \delta$ are roughly able to be done in the same amount of precision as that of x_0.

The precision required for computing the approximation is thus determined by the precision needed for computing the expression inside the square root when computing y from x. This expression is further dominated by its first term $(b^2-4ac)x^2$. Assume that the magnitude of L is about the same as the magnitude of M. Then b^2-4ac can be computed in $2\log_2 M+3$ amount of precision and x^2 can be computed in roughly $2(\log_2 M-\log_2 \epsilon)$ amount of precision. So we need roughly $4\log_2 M-2\log_2 \epsilon+4$ amount of precision in total for this purpose.

After comparing the precision need for classification with the precision need for computing an approximation, we obtain

Theorem 5.9 Assume that the set of points after applying piecewise linear approximation method is in the region of $[-M,M]\times[-M,M]$ and the second order derivative of y with respect to x is bounded in $[-M,M]\times[-M,M]$, then the point can be classified with respect to the conic in $4(\log_2 M-\log_2 \epsilon+1)$ amount of precision where ϵ is a given tolerance.

5.4.2 Point/Quadric Classification

Analogous to the point/conic classification, we assume that each quadric surface has been triangularized by a set of points in the 3-D region $[-M,M]\times[-M,M]\times[-M,M]$. For each point on the surface, there is a corresponding triangular patch formed by three points from the given set such that the distance from the point to the plane that contains the triangle along z direction is within ϵ.

Since the only common points of triangles and surfaces are vertices of triangular patches, there is no immediate error estimation formula available for estimating distance in z direction between an arbitrary point on the surface patch and the corresponding point on the triangle. We assume that coordinates of the points in the given piecewise linear approximation can be represented in N amount of precision. Then, if $P_1=(x_1,y_1,z_1)$, $P_2=(x_2,y_2,z_2)$ and $P_3=(x_3,y_3,z_3)$ are the three end points of a

triangular patch, the equation of the plane passing through P_1, P_2 and P_3 is

$$\begin{vmatrix} x & y & z & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} = 0$$

or equivalently,

$$\begin{vmatrix} y_1 & z_1 & 1 \\ y_2 & z_2 & 1 \\ y_3 & z_3 & 1 \end{vmatrix} x - \begin{vmatrix} x_1 & z_1 & 1 \\ x_2 & z_2 & 1 \\ x_3 & z_3 & 1 \end{vmatrix} y + \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} z - \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} = 0$$

Thus, $2N+2$ and $3N+3$ amount of precisions are necessary to represent the linear term coefficient and constant term of the plane equation respectively. By the result from the point/plane classification, therefore, we can classify a point with respect to a quadric surface in $5(N+2)$ amount of precision using piecewise linear triangular approximation.

5.4.3 Remark

One can see from the above analysis that once any curve has been approximated by a piecewise linear approximant, then the point can be classified with respect to the corresponding curve in roughly four times the amount of precision used to represent the coordinates of the approximating points. However, the precision needed to compute the approximation may exceed the precision required for the point/line classification. Therefore, the actual amount of precision required for the approximated point/curve classification also depends on the precision of the approximating points and the precision required for computing the piecewise linear approximation.

This principle extends equally well to surfaces that are approximated by piecewise linear triangular patches. The amount of precision required for the point/surface classification again is the larger of five times the precision for representing the coordinates of the approximating points, and the precision required for computing the piecewise linear triangular approximation.

6. CONCLUSIONS

For the past decade, computational geometry has been one of the fastest growing fields attracting many researchers to design efficient algorithms to deal with constructing, manipulating and querying geometric objects. Now, with many experiences of actual implementations of such algorithms using floating point arithmetic and the lessons of possible failures resulting from such a computational model, we have seen a gap between designing geometric algorithms and implementing them on a computer. The gap is between the ideal assumption of infinite precision when algorithms are designed and the practical implementation of finite precision when they are implemented using floating point arithmetic on a computer. We call it the robustness problem in geometric computation.

The objective of this thesis has been to study the problem of when and where failures of geometric algorithms occur. Because of the diversity of different classes of geometric algorithms and geometric objects, we have to restrict ourselves to a certain class of geometric algorithms and geometric objects when studying the problem. The class of geometric algorithms and objects we have chosen are Boolean operations and nonmanifold polyhedra. We have chosen Boolean operations because they are important in solid modeling and we have chosen nonmanifold polyhedra because they are closed under Boolean operations. We have taken the approach of designing our own Boolean intersection algorithm and building our own solid modeler and have implemented our algorithm. We have incorporated both floating point arithmetic and exact rational arithmetic into our modeler. The program decides which mode of arithmetic to choose depending on the input vertex coordinates. Having two arithmetic modes enables us to make comparison between the results of Boolean operations on two identical set of input objects with vertex coordinates of the first set in rational

numbers and vertex coordinates of the second set in floating point numbers. Experimenting with our modeler, we have been able to find cases of failure of floating point arithmetic for certain specially positioned input solids. After analyzing these failures, we have concluded that the reason for such failures is that the algorithm made inconsistent decisions, based on numerical computations, about incidence structures.

To circumvent the inherent difficulty associated with the model of floating point arithmetic when representing geometric object, we next have taken an approach similar to that originally proposed by Sugihara and Iri [33] and tried to apply exact rational arithmetic in geometric computation. We have restricted ourselves to the problem of estimating precision requirements for point/line, point/plane, point/conic and point/quadric classifications, since these classifications are some of the most fundamental geometric operations used in implementing Boolean operations. We have shown that the point/line and point/plane classifications can be done precisely with only a moderate increase of the input precision. However, similar results obtained for point/conic and point/quadric classifications are very unfavorable for doing exact classification. So, we have provided an estimate for point/conic and point/quadric classification when conics and quadrics are piecewise linearly approximated.

The major contributions of the thesis are the following:

We have provided a data structure for representing nonmanifold polyhedra and a corresponding Boolean intersection algorithm for manipulating the representations. In our representation, the emphasis is put on how to store and transfer intersection information so that multiple computations of the same numerical quantity can be avoided. Although conceptually our Boolean intersection algorithm is the same as some other Boolean intersection algorithms, especially the one by Hoffmann [12], the paradigm of our algorithm to traverse out intersection shells after intersection analysis is new. Using this traversing paradigm, we do not need to create a new edge segment whenever an edge is subdivided by an intersection point. The algorithm only creates edges that belong to the intersection solid.

We have claimed an expression on the minimum distances from a point to a line in 2-D and a point to a plane in 3-D. The solution to the problems may have its own merit from a mathematical point of view. It at least provides us with a method for proving lower bounds on the precision required for the point/line and point/plane classifications. When solving the point/conic and point/quadric classification problems, we have developed a resultant based method. The method is, in fact, a general one which can be applied to classification problems in a similar setting. From our results on the precision required for classification problems, we have drawn the conclusion that exact methods may be useful in modeling activities, such as implementing Boolean operations, but only when dealing with objects in a linear domain such as lines and planes. Moreover, exact methods are not practically useful when dealing with nonlinear, curved objects. Finally, we have given a general precision estimate for point/conic and point/quadric classifications when conics and quadrics are piecewise linearly approximated.

Compare with the pace of new geometric algorithms being designed, the pace of research on the robustness issue in geometric computation is slow. Two factors may be contributing to this: First, and most importantly, the program implementing a geometric algorithm works most of the time in everyday use. Because failures happen infrequently, users tend to tolerate and forgive program failures that are caused by two indistinguishable incidence structures. With no pressing need placed on the robustness issue, the general perception has been that it is more important to design new geometric algorithms than to improve or redesign existing algorithms so that they are more robust. Secondly and more fundamentally, when representing and manipulating geometric objects by a computer, we actually make a compromise in using the finite arithmetic of the computer to approximate a continuous domain of possible geometric objects. Numerical analysis studies the effect of computation on approximated numerical values. However, it does not provide a solution for how to make consistent topological decisions based on approximated numerical values, a process common in geometric computation. The lack of theoretical foundations to

guide proper topological decision making is the major obstacle in the way of solving the robustness problem. There are attempts by some researchers to formalize computational models that are suitable for geometric computation in the presence of floating point arithmetic [14, 15, 16, 2]. But, much more work needs to be done before such a complete theory can be established. We feel that a provably robust Boolean algorithm using floating point arithmetic is still in the distant future.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] B. G. Baumgart. Winged-Edge Polyhedron Representation. Technical Report CS-320, AI Lab, Stanford University, 1972.

[2] B. Brüderlin. Detecting Ambiguities: An Optimistic Approach to Robustness Problems in Computational Geometry. Manuscript.

[3] H. Chiyokura. *Solid Modeling with Designbase.* Addison-Wesley, 1988.

[4] D. Dobkin and D. Silver. Recipes for Geometry and Numerical Analysis, Part I: A Empirical Study. In *4th ACM Symposium on Computational Geometry*, pages 93–105, Urbana, Illinois, June 1988. ACM.

[5] A. Dresden. *Solid Analytical Geometry and Determinants.* John Wiley & Sons, 1930.

[6] I. D. Faux and M. J. Pratt. *Computational Geometry for Design and Manufacture.* Ellis Horwood, 1979.

[7] J. Flaquer, A Carbajal, and M A Mendez. Edge-Edge Relationships in Geometric Modelling. *Computer-Aided Design*, 1987.

[8] J. D. Foley and A. Van Dam. *Principles of Interactive Computer Graphics.* Addison-Wesley, second edition, 1982.

[9] L. Guibas, D. Salesin, and J. Stolfi. Epsilon Geometry: Building Robust Algorithms from Imprecise Computations. In *Proceedings of the 5th Annual ACM Symposium on Computational Geometry.* ACM, 1989.

[10] L. Guibas and J. Stolfi. Primitives for the Manipulation of General Subdivisions and the Computations of Voronoi Diagram. *ACM Transactions on Graphics*, pages 74–123, April, 1985.

[11] L. E. Heindel. Integer Arithmetic Algorithms for Polynomial Real Zero Determination. *Journal of the ACM*, 18(4), 1971.

[12] C. M. Hoffmann. *Geometric and Solid Modeling – An Introduction.* Morgan Kaufmann, San Mateo, California, 1989.

[13] C. M. Hoffmann. The Problems of Accuracy and Robustness in Geometric Computation. *IEEE Computer*, 22(3), March 1989.

[14] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick. Robust Set Operations on Polyhedral Solids. Technical Report 723, Computer Sciences Department, Purdue University, W. Lafayette, IN 47907, 1987.

[15] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick. Towards Implementing Robust Geometric Computations. In *4th ACM Symposium on Computational Geometry*, pages 106–117, Urbana, Illinois, June 1988. ACM.

[16] J. E. Hopcroft and P. J. Kahn. A Paradigm for Robust Geometric Algorithms. Technical Report TR 89-1044, Department of Computer Science, Cornell University, 1989.

[17] M. Karasick. *On the Representation and Manipulation of Rigid Solids.* PhD thesis, Computer Science Department, McGill University, Montreal, Canada, 1988.

[18] J. Levin. Mathematical Models for Detecting the Intersections of Quadric Surfaces. *Computer Graphics and Image Processing*, 11:73–87, 1979.

[19] M. Mäntylä. Boolean Operations of 2-Manifolds Through Vertex Neighborhood Classification. *ACM Transaction on Graphics*, 5:1–29, 1986.

[20] M. Mäntylä. *An Introduction to Solid Modeling.* Computer Science Press, Rockville, Maryland, 1988.

[21] M. Migotte. Some Useful Bounds. In B. Buchberger, G. E. Collins, and R. Loos, editors, *Computer Algebra — Symbolic and Algebraic Computation*, pages 259–263. Springer-Verlag, 1983.

[22] V. Milenkovic. Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic. Technical Report CMU-CS-88-168, Computer Science Department, Carnegie Mellon University, 1988.

[23] W. M. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, second edition, 1979.

[24] A. Paoluzzi, M. Ramella, and A. Santarelli. Un modellatori geometrico su rappresentazioni triango-alate. Technical Report TR 13.86, Department of Information and Systems, University of Rome, Italy, 1986.

[25] L. Piegl. Representation of Quadric Primitives by Rational Polynomials. *Computed Aided Geometric Design*, 2:151–155. 1985.

[26] L. Piegl. The Sphere as a Rational Bézier Surface. *Computer Aided Geometric Design*, 3:45–52, 1986.

[27] L. Piegl. Geometric Method of Intersecting Natural Quadrics Represented in Trimmed Surface Form. *Computered-Aided Design*, 21(4):201–212, May 1989.

[28] L. Piegl and W. Tiller. Curve and Surface Constructions using Rational B-Splines. *Computer-Aided Design*, 19(9):485–498, November 1987.

[29] A. A. G. Requicha and H. B. Voelcker. Solid Modeling: A Historical Summary and Contemporary Assessment. *IEEE Computer Graphics and Applications*, 2:9–24, 1982.

[30] A. A. G. Requicha and H. B. Voelcker. Solid Modeling: Current Status and Research Directions. *IEEE Computer Graphics and Applications*, 3:25–37, 1983.

[31] A. A. G. Requicha and H. B. Voelcker. Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms. In *Proceedings of IEEE 73*, pages 30–44. IEEE, 1985.

[32] M. Segal and C. H. Séquin. Consistent Calculations for Solid Modeling. In *Proceedings of the Symposium on Computational Geometry*, pages 29–37. ACM, 1985.

[33] K. Sugihara and M. Iri. A Solid Modelling System Free from Topological Inconsistency. *Information Processing*, 12(4):380–393, 1989.

[34] K. Sugihara and M. Iri. Construction of the Voronoi Diagram for over 10^5 Generators in Single-Precision Arithmetic. In *Proceedings of First Canadian Conference on Computational Geometry*. Montreal, Canada, August 21-25 1989.

[35] W. Tiller. Rational B-Splines for Curve and Surface Representaion. *IEEE Computer Graphics and Applications*, 3(6):61–69, September 1983.

[36] B. L. van der Waerden. *Modern Algebra*, volume I. Springer Verlag, 1967.

[37] G. Vaněček. *Set Operations on Polyhedra using Decomposition Methods*. PhD thesis, Computer Science Department, University of Maryland, College Park, Maryland, 1989.

[38] K. J. Weiler. *Topological Structures for Geometric Modeling*. PhD thesis, Computer and System Engineering Department, Rensselaer Polytechnic Institute, 1986.

[39] Kiyonori Yosida. On Computational Accuracy in Determining the Sign of an Algebraic Quantity. *Electronics and Communications in Japan*, Vol. J69(No. 5), 1986. (in Japanese).

VITA

VITA

Jiaxun Yu was born on February 23, 1963 in Shanghai, China. In 1980, after graduating from the Shanghai High School in Shanghai, China, he continued his education at Shanghai Jiao Tong University in Shanghai, China where he received his B.S. degree majoring in Computer Science in 1984. After graduation, he joined the faculty of the Department of Computer Science and Engineering of the Shanghai Jiao Tong University as a teaching staff. In August 1985, Mr. Yu began his graduate studies in Computer Science at Purdue University in West Lafayette, Indiana where he received his M.S. and Ph.D. degrees in Computer Science in 1988 and 1991, respectively. While at Purdue, Mr. Yu was supported as a research assistant with the geometric modeling group of the Computing About Physical Objects project under the supervision of Professor Christoph Hoffmann.